Cyberlogic®

Math & Logic Help

Math & Logic Engine
for OPC Servers

Version 9

MATH & LOGIC HELP

For Cyberlogic OPC Servers

Version 9

Document last revision date May 5, 2017

TABLE OF CONTENTS

INTRODUCTION

The Cyberlogic OPC Server provides OPC Data Access, Alarms & Events and XML Data Access functions. Its modular structure supports a variety of industrial devices and communication networks. As a result, the server maintains a set of common features, but has the flexibility to allow the addition of optional features as required for specific applications.

Math & Logic is one of these optional features. With it, you can create mathematical or logical functions that can operate on any data items that are available to the OPC server. The results of the functions are available as OPC data items to any attached OPC client. The full-featured Math & Logic is included with the following Cyberlogic products:

- DHX OPC Premier Suite
- MBX OPC Premier Suite
- OPC Crosslink Premier Suite
- OPC Datacenter Premier Suite
- DHX OPC Enterprise Suite
- MBX OPC Enterprise Suite
- OPC Crosslink Enterprise Suite

Note This document includes only the information that is specific to the Math & Logic feature. For information on the common features of the Cyberlogic OPC Server, refer to the Cyberlogic OPC Server Help.

Pre-Programmed and Demo Math & Logic Items

The Cyberlogic OPC products that do not include the full-featured Math & Logic do have a limited Math & Logic capability. There are two parts to this:

You can configure Math & Logic data items that use pre-programmed trigger and switch applets. With these applets, you don't write any programs. You simply configure the applet through a dialog, and the program code is then automatically generated. The generated code cannot be modified, but it will run unrestricted. You will find these applets very useful as enable or trigger signals.

You can also create Math & Logic data items that use custom programs, but you can run them only in demo mode. This means that they will run only for the first two weeks after the software is installed, and after that will run only for two hours after the system is re-booted. This allows you to try out the Math & Logic feature, but you should not use these demo-mode programs in your application.

Note The names of Math & Logic data items that run in demo mode are shown in the Address Space tree in a different font color, to distinguish them from data items that are fully functional. In most systems, they will use a blue font instead of the black font that other data items use. However, these colors are taken from the Windows system colors and may be different on your system.

The Math & Logic Program Types section has complete information on the pre-programmed triggers and switches.

Compatibility and Compliance

Math & Logic is a feature of Cyberlogic's family of OPC products. The Cyberlogic OPC Server can provide data to any OPC Foundation compliant client from any supplier.

Cyberlogic OPC products provide full compliance with the OPC Foundation specifications for:

- Data Access 3.0, 2.05a and 1.0a
- Alarms & Events 1.1
- XML Data Access 1.0
- Data Access Automation 2.02

These products are tested for compliance to the OPC specifications using the latest test software from the OPC Foundation. All Cyberlogic OPC products are certified for compliance by the OPC Foundation's Independent Testing Laboratory. In addition, they are tested annually for interoperability with other OPC products at the OPC Foundation's Interoperability Workshops.

WHAT SHOULD I DO NEXT?

The links below will take you directly to the section of this manual that contains the information you need to configure, use and troubleshoot Math & Logic.

This document describes only the features specific to Math & Logic. For information on the common features of the Cyberlogic OPC Server, refer to the Cyberlogic OPC Server Help.

Learn How Math & Logic Works

If you are not familiar with the way that Math & Logic performs its operations and provides results to OPC clients, you should begin by reading the Theory of Operation.

Read a Quick-Start Guide

First-time users of Math & Logic will want to refer to the Quick Start Guide for a step-by-step walk through a typical configuration session.

Get Detailed Information on the Configuration Editors

Experienced users who want specific information on features of the configuration editors will find it in the Configuration Editor Reference section.

Verify That It's Working or Troubleshoot a Problem

If you have already configured the server, you should verify that it operates as expected. Refer to the Validation and Troubleshooting section for assistance. In case of runtime problems, this section also provides problem-solving hints.

Print a Copy of This Document

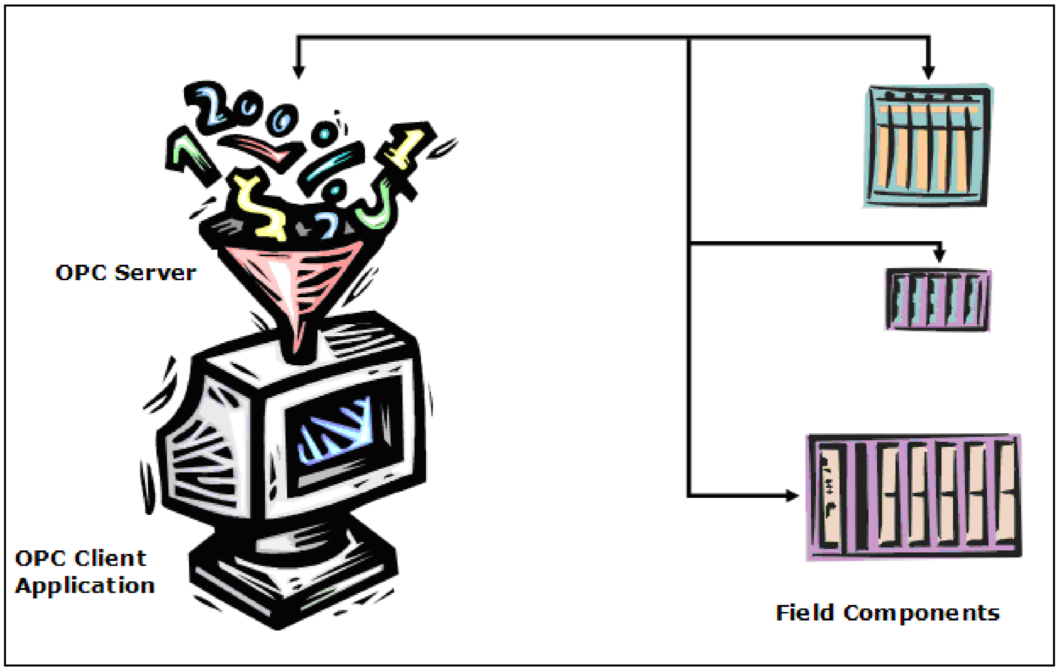
The content of this document is also provided in PDF format. Use the Adobe® Reader program to view and print the PDF file.

Contact Technical Support

To obtain support information, open the Windows ***Start*** menu and go to ***Cyberlogic Suites***, and then select ***Product Information***.

THEORY OF OPERATION

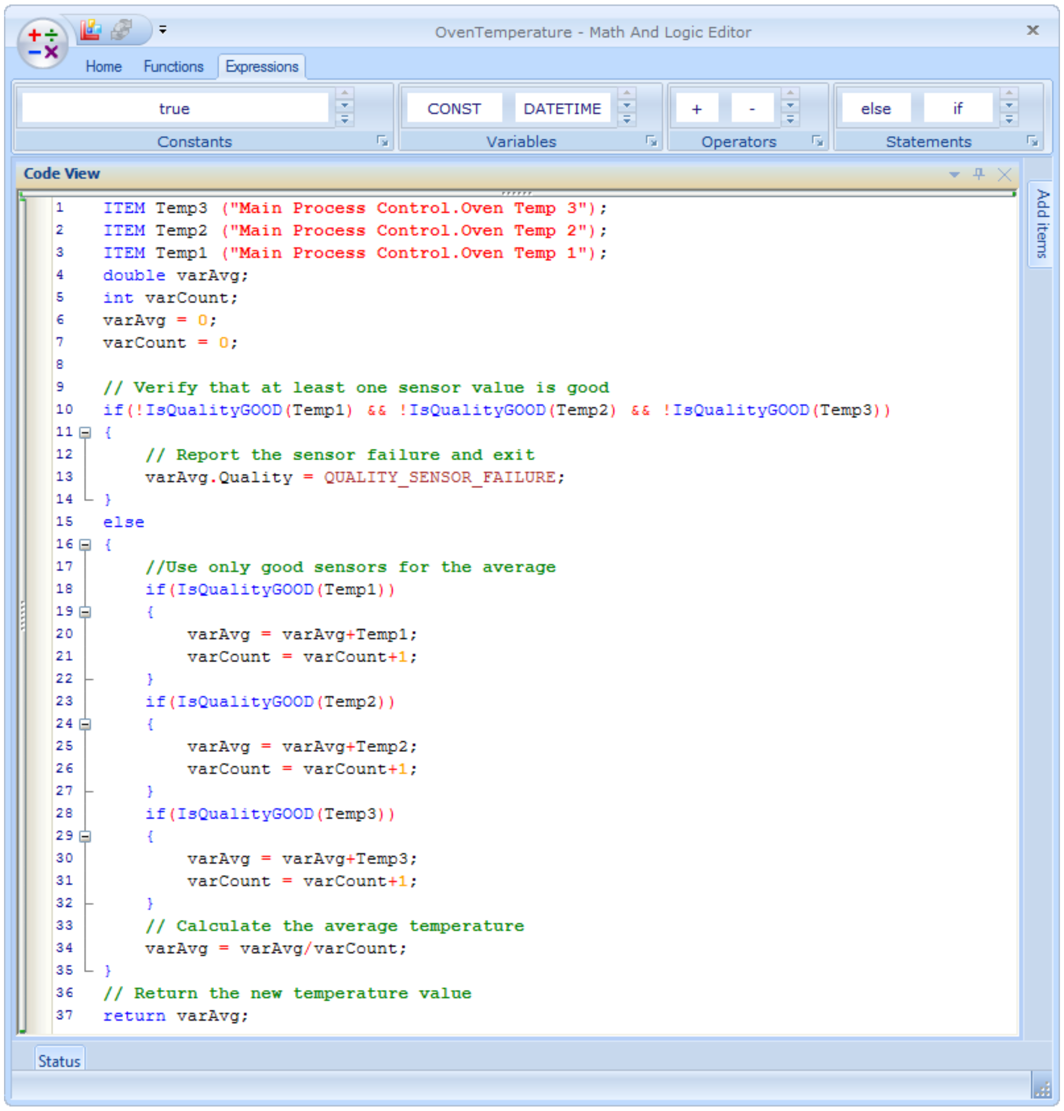
This section will familiarize you with the main features of the Cyberlogic OPC Server as they relate to Math & Logic. Refer to the Cyberlogic OPC Server Help for a full discussion of the common features of the Cyberlogic OPC Server. If you are new to OPC or the Cyberlogic OPC Server, we strongly recommend that you read the OPC Tutorial first. You will find it in the Help section of your product installation.



The basic function of an OPC server is illustrated in the figure above. The server obtains data from field components and presents it, in a standard way, to OPC client applications. Typically, these field components are PLCs or similar devices. More advanced servers will also allow you to process the incoming raw data prior to making it available to the client applications.

Math & Logic

Math & Logic is an optional feature for Cyberlogic's OPC Server, which allows you to write programs in the C-logic™ programming language. These programs can include complex mathematical and logical functions that operate on one or more data items. The results appear as data items that are available to the OPC clients.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;
varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{
    // Report the sensor failure and exit
    varAvg.Quality = QUALITY_SENSOR_FAILURE;
}
else
{
    //Use only good sensors for the average
    if(IsQualityGOOD(Temp1))
    {
        varAvg = varAvg+Temp1;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp2))
    {
        varAvg = varAvg+Temp2;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp3))
    {
        varAvg = varAvg+Temp3;
        varCount = varCount+1;
    }
    // Calculate the average temperature
    varAvg = varAvg/varCount;
}
// Return the new temperature value
return varAvg;
```

A typical program associated with a Math & Logic data item

Advantages of Server-Based Math & Logic

Basic OPC servers get data from field devices, such as PLCs, and present it to OPC client software. They may provide a limited ability to convert the values into engineering units, and may also have some ability to simulate the data. If you need to do anything more complex than those simple functions, the client software will have to handle it.

That may be acceptable for simple systems. But if your needs require you to program numerous clients to perform the same calculations, that approach quickly becomes inefficient and error-prone. Furthermore, it limits your choice of client software to high-end packages that include the needed Math & Logic functions.

A far better approach is to use Cyberlogic's Math & Logic to process the data in the OPC server. This has several advantages:

- You program the calculations and logical functions once for the entire system.

- The programs are located and maintained on a single system for easy maintenance. All clients will receive the same data.
- A single Math & Logic engine for all of your users means that you have only one editor and one programming language to learn.
- You will not need to purchase client software to perform the Math & Logic functions.
- Your client software options are not limited to only those packages that include Math & Logic functions.

Math & Logic Applications

Math & Logic provides you with a general programming language, C-logic, that gives you the freedom to manipulate the data as you wish. This opens many possibilities.

Conversion

Most OPC servers have simple linear and square-root conversion functions built-in. With Math & Logic, you can create much more complex conversions that involve:

- Logarithmic, exponential, stepwise or other types of functions
- Clipping and range limiting
- Conversions that can be controlled by data items, allowing you to adjust their parameters in real time

Simulation

Not only can you generate a nearly limitless variety of waveforms, but you have a broad range of control over them. Possibilities include:

- Two sine waves, with an adjustable phase difference
- Two square waves in quadrature to each other, to simulate an encoder

Alarms

Instead of simple alarms that trigger when a value is outside of predetermined limits, you can program alarms for much more complex scenarios:

- A process with temperature requirements that vary depending on the pressure, so you want an alarm when the combination is out of limits
- A tank's fluid level is dropping too rapidly, even if it is still within limits
- A flow rate with limits that vary depending on the current step in the process

Smoothing and Averaging

You can write programs that will minimize the effect of inconsistencies in sensor readings or other measurements:

- Average the readings of two pressure transducers

- Maintain a moving average of the last ten cycle times
- Apply exponential smoothing to a temperature reading

Ranking, Choosing and Voting

The comparison and logical functions allow you to program complex decisions:

- Rank four machines in order of their cycle times
- Identify which of three lines is the closest to going down for scheduled maintenance
- Turn off the heater when three out of four probes reach the temperature set point

Interaction with Other Features

You can write programs to control other features of the Cyberlogic OPC Server:

- Trigger Crosslink transfers
- Enable and disable dynamic Access Paths
- Execute other programs

Unique and Advanced Features of Cyberlogic's Math & Logic

Cyberlogic's Math & Logic implementation includes many advanced features, some of which are unique to Cyberlogic.

Editing Features

- Familiar programming syntax, similar to the C programming language
- Program statements are color-coded, using configurable colors
- Browse window for easy insertion of OPC data item variables

Performance and Diagnostics

- Programs are compiled so they can execute in microseconds, far faster than is possible with interpreted code
- User can compile all programs with a single mouse click
- Compiler can detect program errors at compile time
- Status tags available at runtime include: detailed runtime error information, execution time, number of program executions

Power

- Extensive set of preprogrammed functions
- Each program can produce multiple outputs
- A program can dynamically create data items in the address space, which can be used as inputs, outputs or both

- Program execution triggered on input data change, by a trigger item, at a fixed interval, or as scheduled by the program itself
- Group execution criteria can be overridden for individual programs
- Execution can be enabled and disabled dynamically if there are no subscriptions to the data item
- Math & Logic data items behave like all other data items: can be simulated, used with conversions and alarms, and can be used in other Math & Logic programs

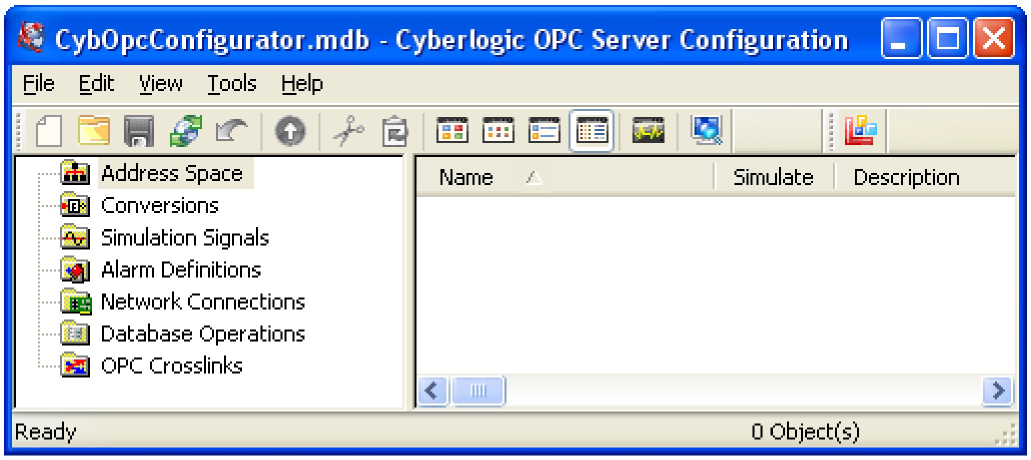
Flexibility

- Broad selection of data types from 1 to 64 bits, including Boolean, signed and unsigned integer, floating point, and string, as well as OPC UA data types
- Support for arrays
- Works with DirectAccess data
- Program inputs can override default deadbands and sampling rates with individual settings

Continue with Main Server Features for a general description of how Math & Logic fits into the server configuration. For an example of how to configure this feature, skip to the Quick Start Guide. Refer to Appendix A: C-logic Language Reference for a complete guide to the C-logic language.

Main Server Features

We will now look at the main features of the Cyberlogic OPC Server, as they relate to Math & Logic.



When you open the Cyberlogic OPC Server Configuration editor, you will find several main trees. These trees represent the main areas that you will configure. Note that some are for premium features that may not be part of the product you have installed, so they will not appear in your configuration. The trees are:

Address Space Tree

This is where you actually configure Math & Logic. To do this, you will create Math & Logic data items and organize these into Math & Logic Devices. You will then write a program for each data item, which will calculate the value that will be assigned to it. For detailed information on how to edit this tree, refer to the Configuration Editor Reference.

Math & Logic Devices

Math & Logic devices group related data items in your configuration. All data items under a device share some common controls and default settings. For example, the device settings specify when the group is enabled, and the default conditions under which the logic will be run. You can choose to enable logic execution, disable it, or use the value of a data item to control the enable and disable. When execution is enabled, you can specify that programs should run when the input data changes, at a fixed interval, or when triggered by a change in the value of a specified data item. Each data item in the device can use these default settings, or define its own.

Math & Logic Data Items

You program the Math & Logic operation in a data item. When the program runs, its result will be available to clients as the value of the data item. It is also possible for a single program to have multiple output values. Refer to Data Item Declarations for more information on how to do this.

Conversions Tree

The Conversions Tree is optional. In it, you can define formulas that can be used to convert raw data values obtained from the field equipment into a form that is more useful to the client. For example, you can change a transducer's voltage value into a pressure value in psi. You can also apply conversions to Math & Logic data items. This allows you to do the calculations on the raw values, and then use a conversion to present the result in engineering units. Refer to the Cyberlogic OPC Server Help for a full discussion of this tree.

Simulation Signals Tree

This tree is also optional. If you want to be able to use simulated data item values instead of real values, you can create various types of simulated data functions in this tree. Simulations are often useful for troubleshooting client applications. You might want to simulate a Math & Logic data item initially, while you work on its program. Refer to the Cyberlogic OPC Server Help for a full discussion of this tree.

Alarm Definitions Tree

You will use this optional tree to interface to OPC Alarms & Events clients. This tree allows you to define the desired alarm conditions and specify what information should be passed as they occur and clear. When you apply these to a Math & Logic data item, the result of the calculation will determine what alarm is triggered. Refer to the Cyberlogic OPC Server Help for a full discussion of this tree.

Network Connections Tree

This tree is where you configure communication to OPC servers, PLCs and other data sources. You will select the networks and interface devices you will use, and configure each of the field components as nodes on those networks. This tree is not directly involved in Math & Logic configuration. However, the network nodes you define here typically provide the input data for the Math & Logic calculations. Refer to the driver agent help files for more information.

Database Operations Tree

The Database Operations Tree is part of the logging feature, which is a premium feature. If this tree is in your product, you can use it to configure databases and data logging operations. Refer to the Data Logger Help for a full discussion of this tree.

OPC Crosslinks Tree

The OPC Crosslinks Tree is part of OPC Crosslink, which is a premium feature. If this tree is in your product, you can use it to configure data transfers between PLCs, between OPC servers and between PLCs and OPC servers. Refer to the OPC Crosslink Help for a full discussion of this tree.

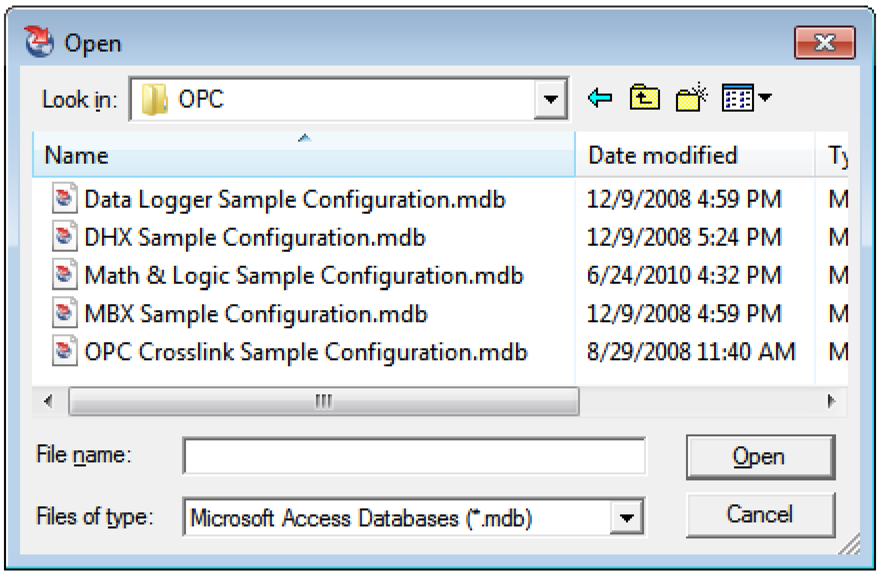
QUICK START GUIDE

Before you can use the OPC server, you must configure it by using the OPC Server Configuration Editor. Math & Logic users must configure the Address Space tree. In addition, users who want to obtain data from PLCs or other OPC servers must configure the Network Connections tree. The remaining trees (Conversions, Simulation Signals, Alarm Definitions and OPC Crosslinks) are optional features used by some systems.

Sample Configuration Files

The default installation of all Cyberlogic OPC Server Suites includes a set of sample configuration files. These samples will help you to understand how to configure the OPC server for your project. In addition, the OPC Math & Logic sample provides you with numerous sample programs that you can modify and use in your system.

To open a sample configuration file from the OPC Server Configuration Editor, open the ***File*** menu and then select ***Open Sample...*** .



Name	Date modified	Ty
Data Logger Sample Configuration.mdb	12/9/2008 4:59 PM	M
DHX Sample Configuration.mdb	12/9/2008 5:24 PM	M
Math & Logic Sample Configuration.mdb	6/24/2010 4:32 PM	M
MBX Sample Configuration.mdb	12/9/2008 4:59 PM	M
OPC Crosslink Sample Configuration.mdb	8/29/2008 11:40 AM	M

A browse window will open to allow you to select the configuration file you want. The available choices will depend on which OPC products you have installed.

The default location of the files is:

C:\Program Files\Common Files\Cyberlogic Shared\OPC.

Step-By-Step Example

This section shows how to configure the Address Space tree for Math & Logic. It does not cover any of the other parts of the configuration. For a step-by-step example of how to configure a complete OPC server, refer to the Cyberlogic OPC Server Help.

You should use this example only as a guideline of how to configure the most common features. For detailed information on all of the Math & Logic features, refer to the

Configuration Editor Reference. The software also includes a configuration file with a set of sample programs. These will help you to understand how the various functions work. They will also give you ideas about what you can do with C-logic, and they can be modified and used in creating your own programs.

For this example, we will assume we have a process for which a temperature measurement is critical. We have three temperature transducers and want to report the average of their readings. However, if a transducer fails, we will average only the values from the good transducers.

The procedure is divided into several sections:

- Creating a Math & Logic Device
- Creating a Math & Logic Data Item
- Editing the Math & Logic Program
- Saving the Configuration and Updating the Server
- Verifying Your Configuration

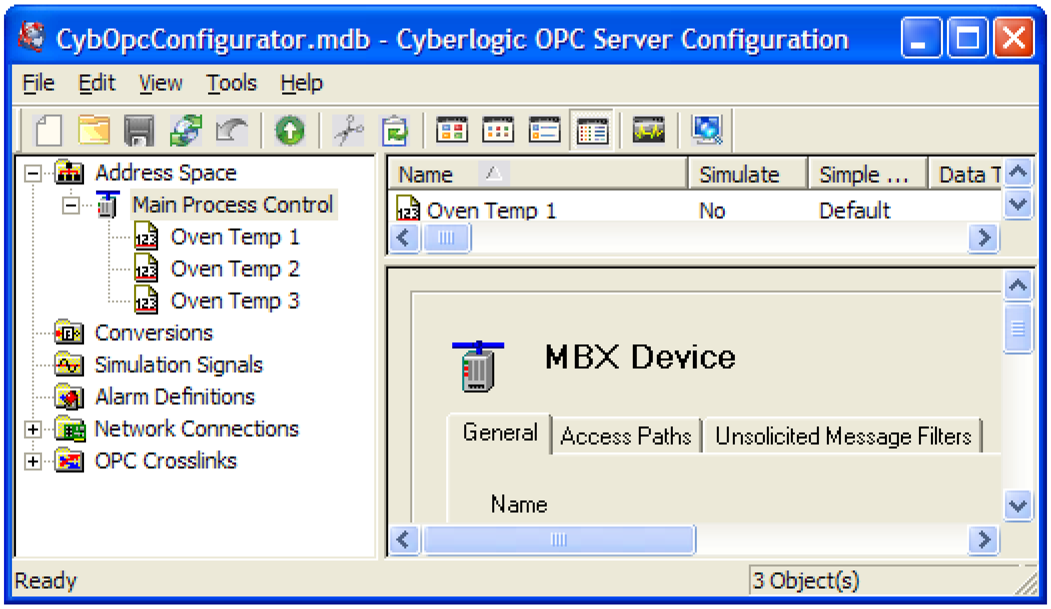
We will begin with Creating a Math & Logic Device.

Creating a Math & Logic Device

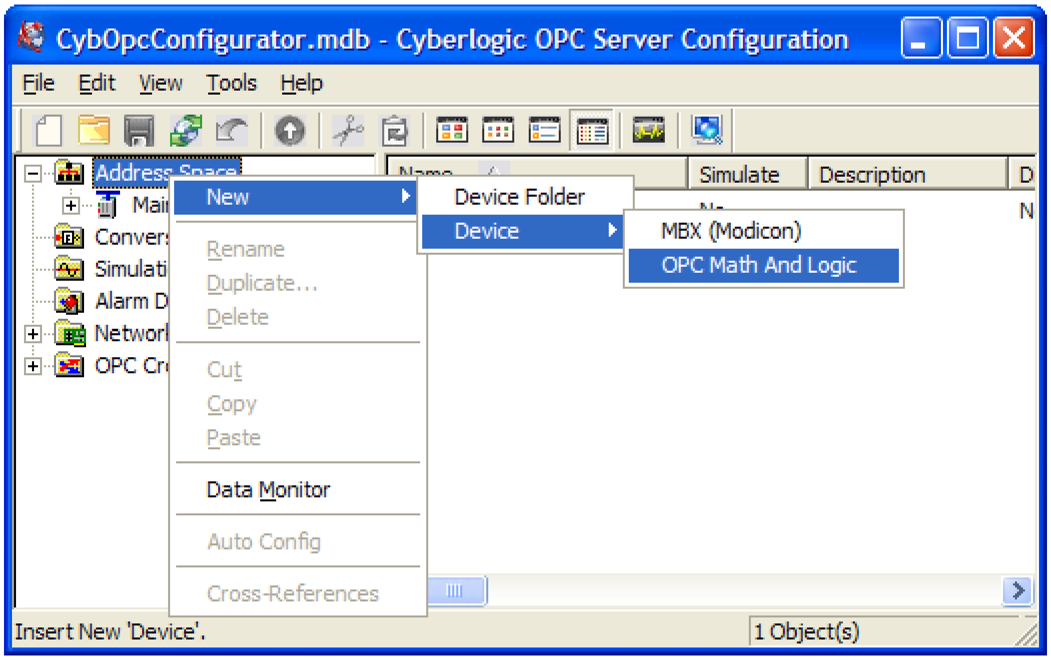
The Math & Logic device groups Math & Logic data items that share some controls and default settings. For example, a device allows you to enable or disable its data items, or to specify a data item that will control the enable status. It also allows you to specify the default conditions under which the data items' programs will execute.

1. To start the editor from the Windows ***Start*** menu, go to ***Cyberlogic Suites***, then open the ***Configuration*** sub-menu, and then select ***OPC Server***.

   If you are running the Cyberlogic OPC Server Configuration Editor for the first time, the editor will prompt you for a configuration file. Click the dialog box's ***Create New...*** button to start with an empty configuration.

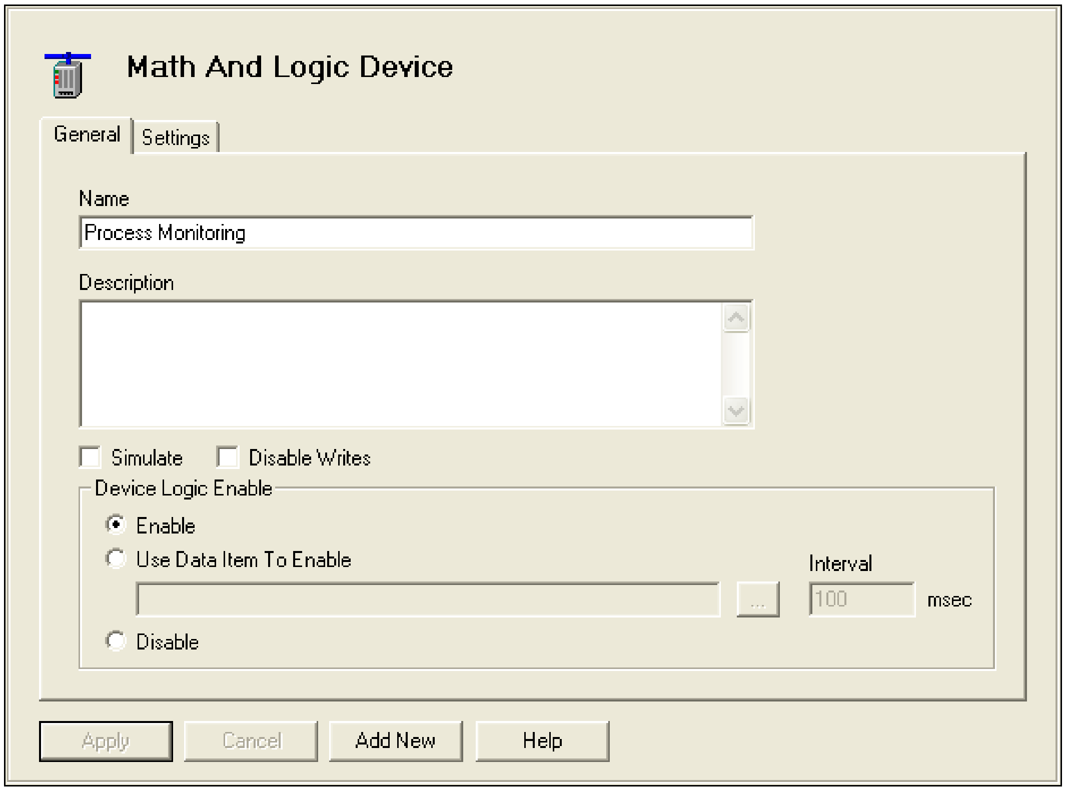


You will see the above screen. For this exercise, we will assume that the three temperature sensors have already been configured. If you want to read about how to do that, refer to the Cyberlogic OPC Server Help.



2. Right-click on the ***Address Space*** tree, then select ***New*** from the context menu, and then ***Device*** and finally ***Math & Logic***.

   The editor will create a device and open it for editing.



Math And Logic Device

General | Settings

Name: Process Monitoring

Description:

☐ Simulate ☐ Disable Writes

Device Logic Enable

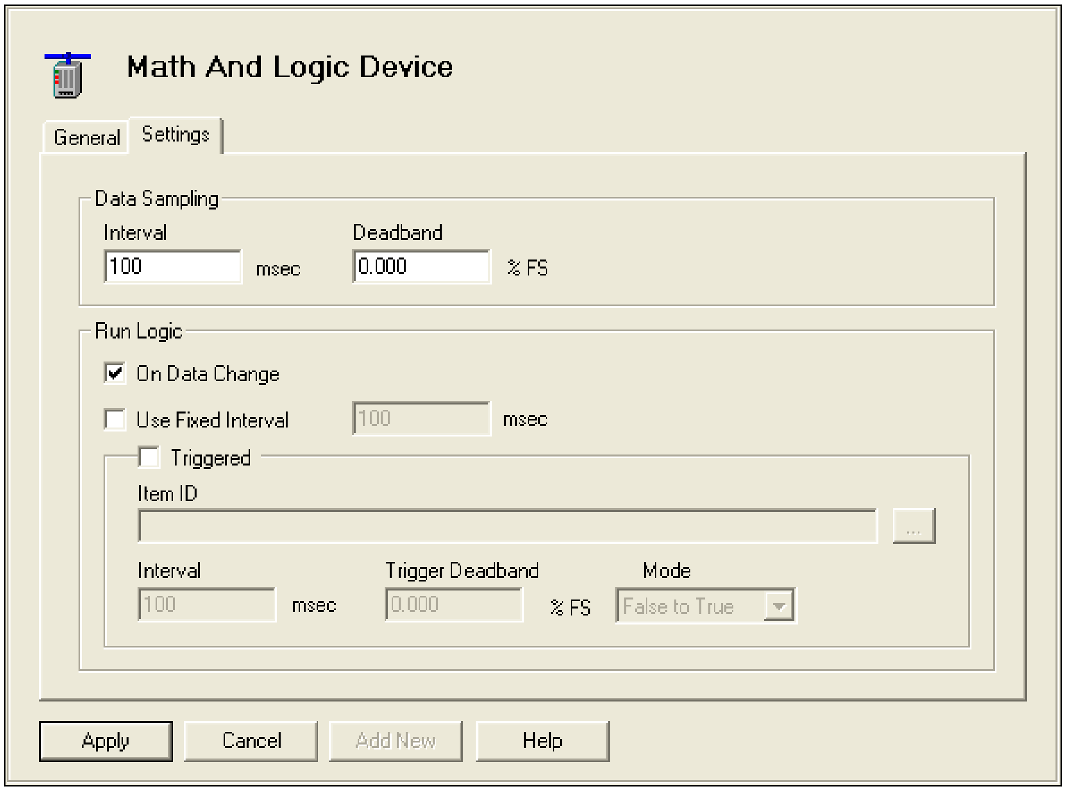
● Enable
○ Use Data Item To Enable
Interval: 100 msec
○ Disable

3. Select the ***General*** tab and enter ***Process Monitoring*** in the ***Name*** field.

4. In the ***Device Logic Enable*** section, select ***Enable***.

   This will keep the programs for all of the data items in the device enabled to run.

5. Select the ***Settings*** tab.



Math And Logic Device

General | Settings

Data Sampling
Interval: 100 msec
Deadband: 0.000 % FS

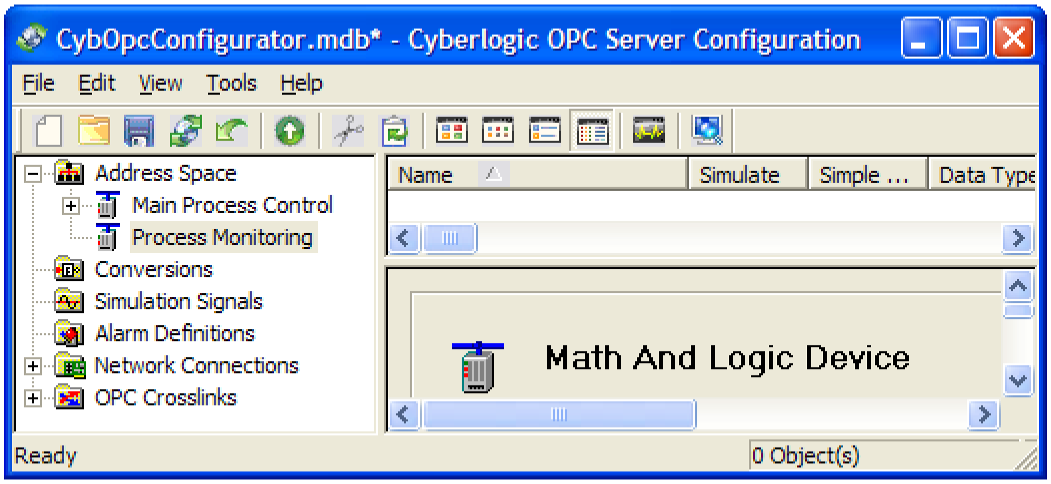
Run Logic
[✓] On Data Change
[ ] Use Fixed Interval 100 msec
[ ] Triggered
Item ID
Interval: 100 msec
Trigger Deadband: 0.000 % FS
Mode: False to True

6. In the ***Run Logic*** section, check ***On Data Change***.

   This will cause the program to calculate the average temperature whenever there is a change in any of the sensor values.

7. Uncheck ***Use Fixed Interval*** and ***Triggered***.

8. Click ***Apply***.

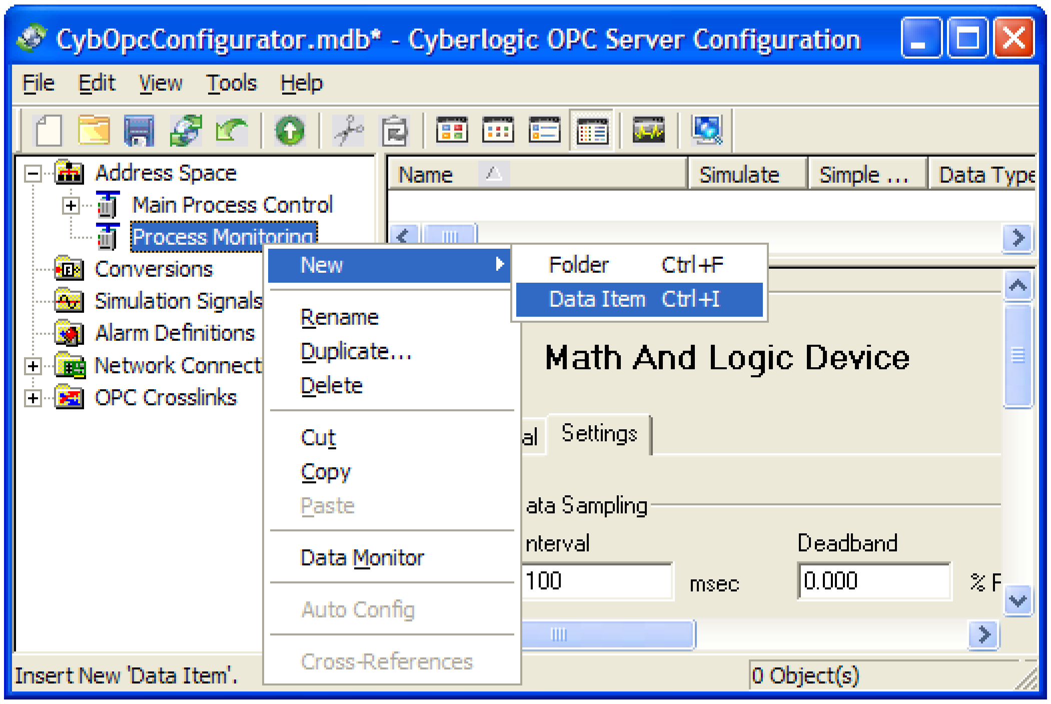


CybOpcConfigurator.mdb* - Cyberlogic OPC Server Configuration

Next, go to Creating a Math & Logic Data Item.

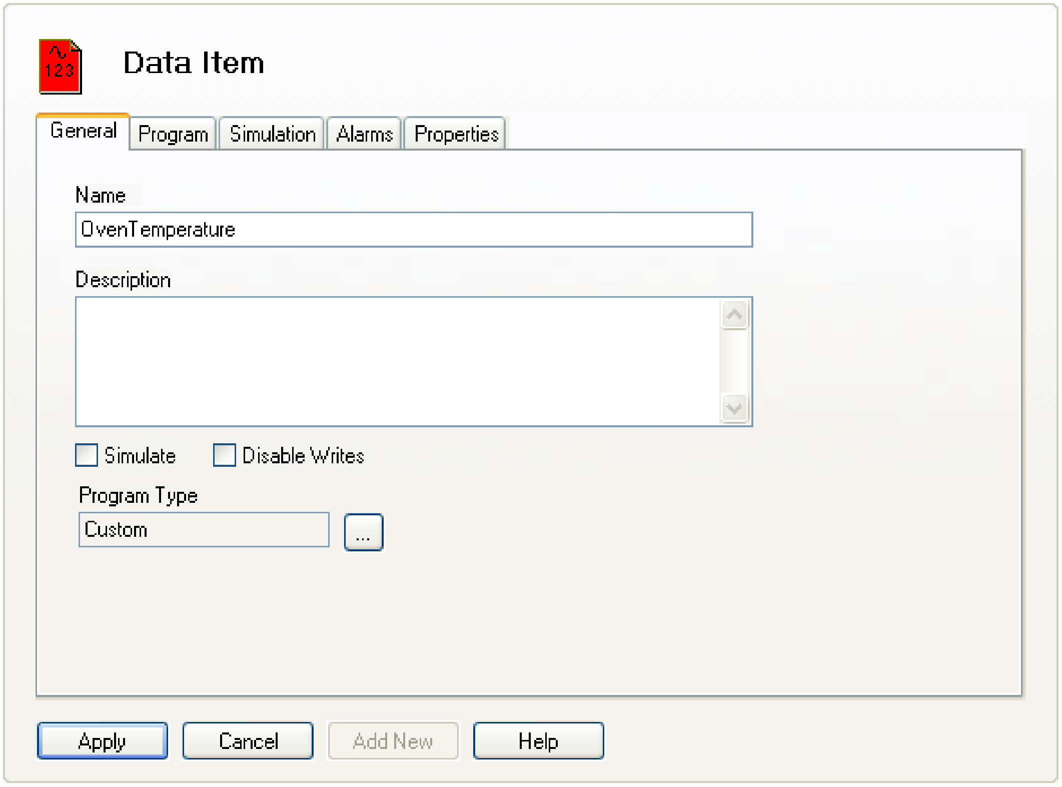
Creating a Math & Logic Data Item

The Math & Logic data item contains the actual program you want to execute, and reports the result as the value of the data item. The data item editor also allows you to modify the enable criteria and override the run criteria for the program.



1. Right-click on the ***Process Monitoring*** Math & Logic device, then select ***New*** from the context menu, and then select ***Data Item***.

   The editor will create a data item and open it for editing.



Name
OvenTemperature

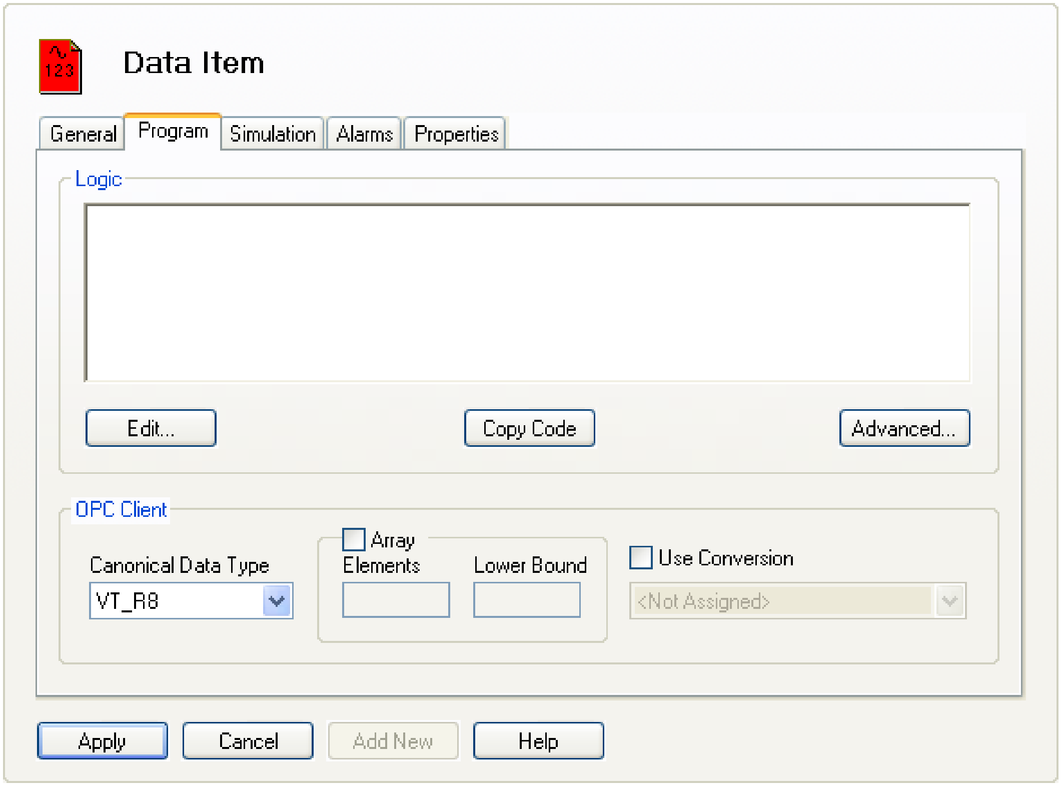
Description

[ ] Simulate [ ] Disable Writes

Program Type
Custom

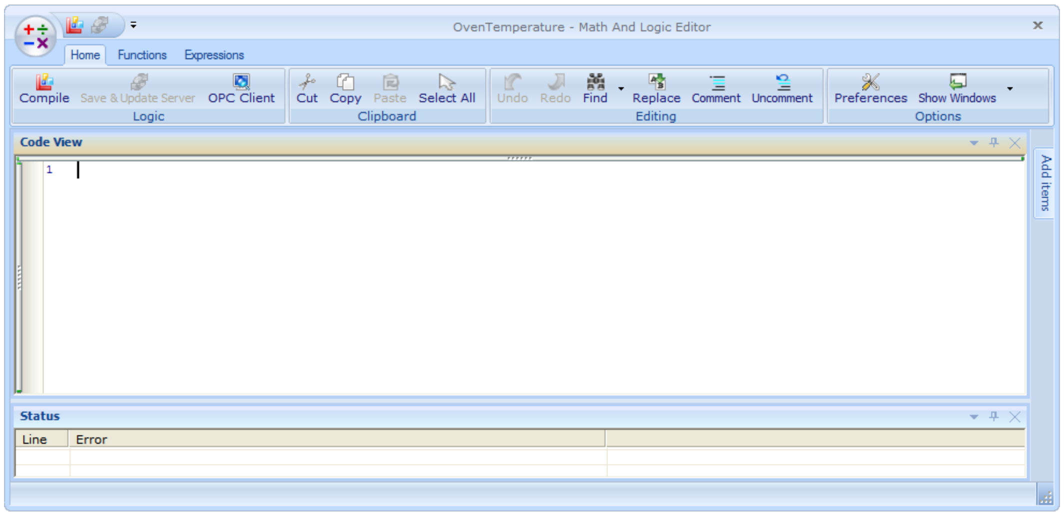
2. Select the ***General*** tab and enter ***OvenTemperature*** in the ***Name*** field.

   Notice the red icon. This indicates that there is no valid compiled program for this data item.



3. Select the ***Program*** tab, and then click the ***Edit...*** button.

   The Math & Logic Editor will open.



Next, go to Editing the Math & Logic Program.

Editing the Math & Logic Program

The Math & Logic Editor allows you to create, edit and compile your programs using Cyberlogic's C-logic programming language. (Refer to Appendix A: C-logic Language

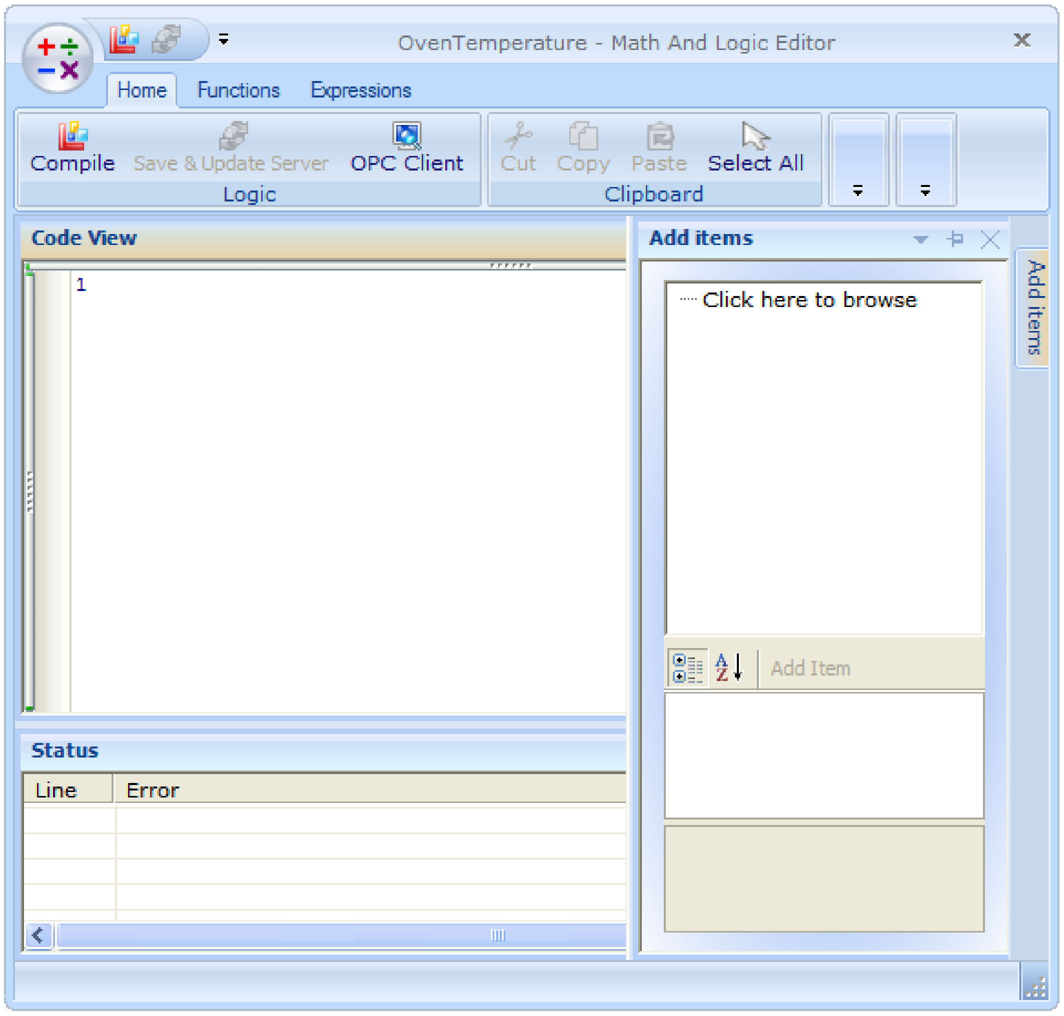
Reference for detailed information on programming in C-logic.) Each program is part of a Math & Logic data item, and so the editor is launched from within the data item.

For this example, we will need three temperature sensor inputs, which we assume have already been configured as data items obtained from a PLC.

We will now create the program shown below.

```
ITEM Temp1 ("Main Process Control.Oven Temp 1");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp3 ("Main Process Control.Oven Temp 3");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(temp1) && !IsQualityGOOD(temp2) &&
!IsQualityGOOD(temp3))
{
      // Report the sensor failure and exit
      varAvg.Quality = QUALITY_SENSOR_FAILURE;
}
else
{
      // Use only good sensors for the average
      if(IsQualityGOOD(Temp1))
      {
            varAvg = varAvg+Temp1;
            varCount = varCount+1;
      }
      if(IsQualityGOOD(Temp2))
      {
            varAvg = varAvg+Temp2;
            varCount = varCount+1;
      }
      if(IsQualityGOOD(Temp3))
      {
            varAvg = varAvg+Temp3;
            varCount = varCount+1;
      }

      // Calculate the average temperature
      varAvg = varAvg/varCount;
}

// Return the new temperature value
return varAvg;
```

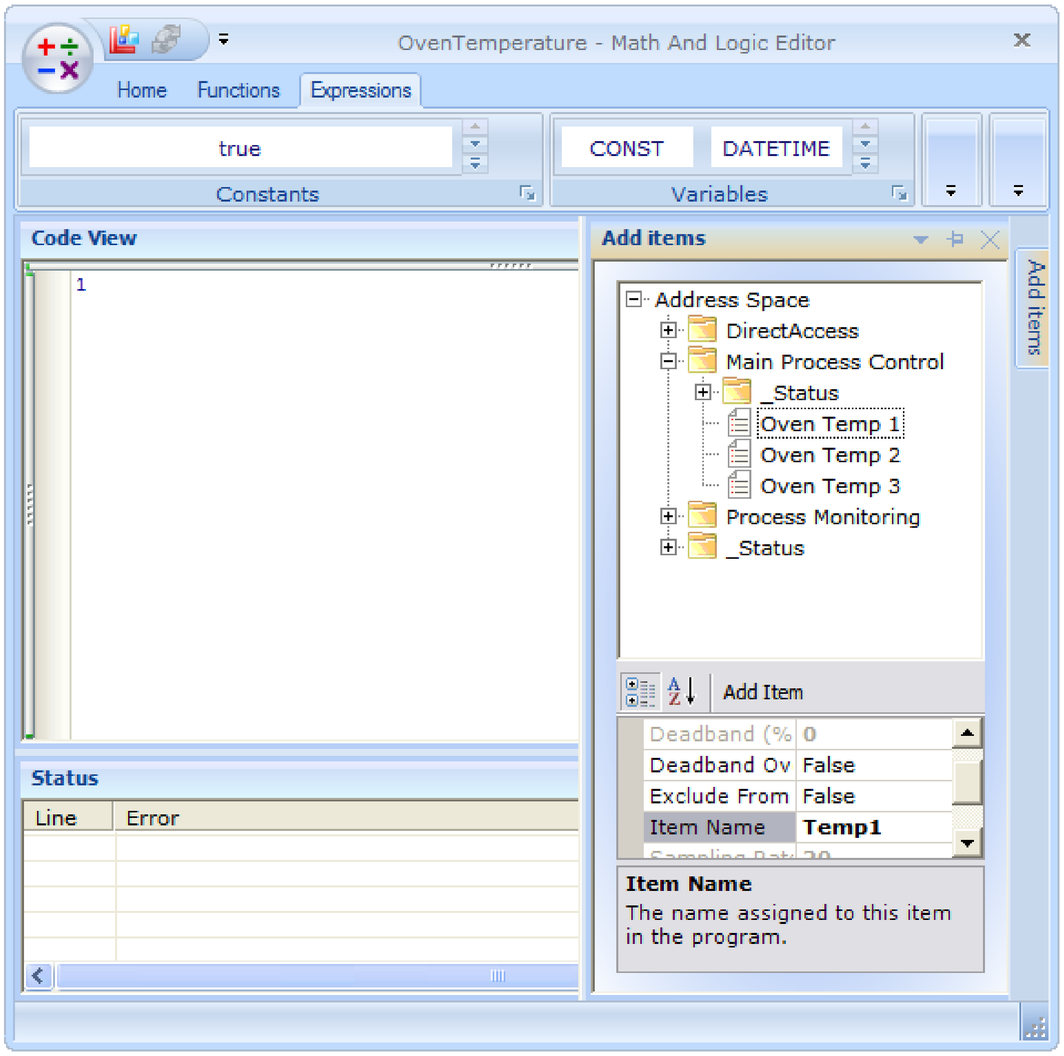
Note It is possible to simply type the program in the Code View pane just as it is shown. Instead, we will use the editor's built-in tools to demonstrate how they can help in creating your program.



1. Go to the ***Expressions*** ribbon, and click on ***Add Items*** to open the Add Items pane.

2. Click on ***Click here to browse***.

   The editor will populate the browser tree with all items in the server's Address Space.



Deadband (%	0
Deadband Ov	False
Exclude From	False
Item Name	Temp1

Item Name
The name assigned to this item in the program.

3. Open the ***Address Space*** items to display a tree showing the available data items. Notice that this includes status and DirectAccess items.

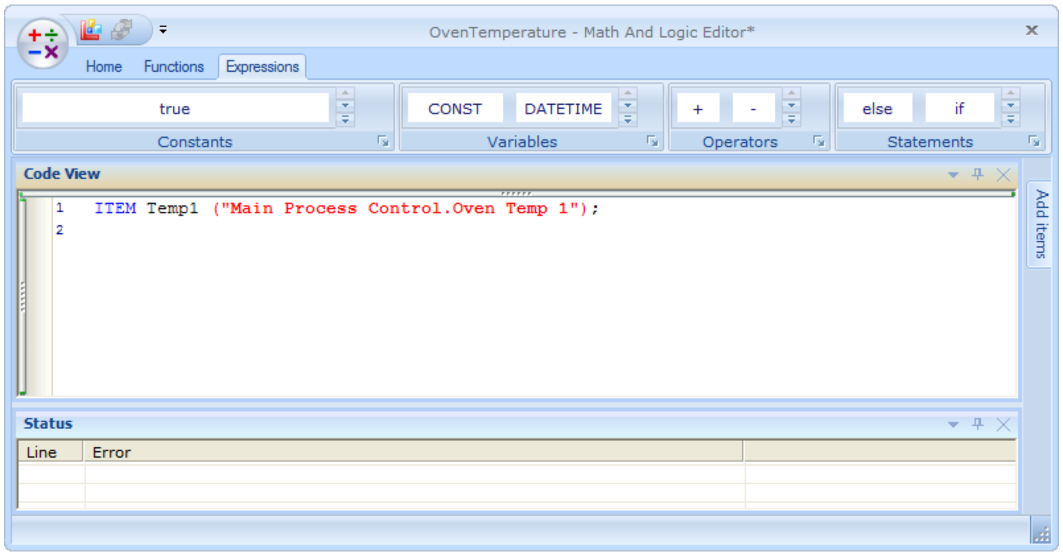
   The data items we need have been configured as Oven Temp 1, Oven Temp 2 and Oven Temp 3 in the Main Process Control PLC.

4. Select ***Oven Temp 1*** to open its OPC Properties editing fields.

5. In the ***Item Name*** field, enter ***Temp1***. This is the name the program will use for this data item.

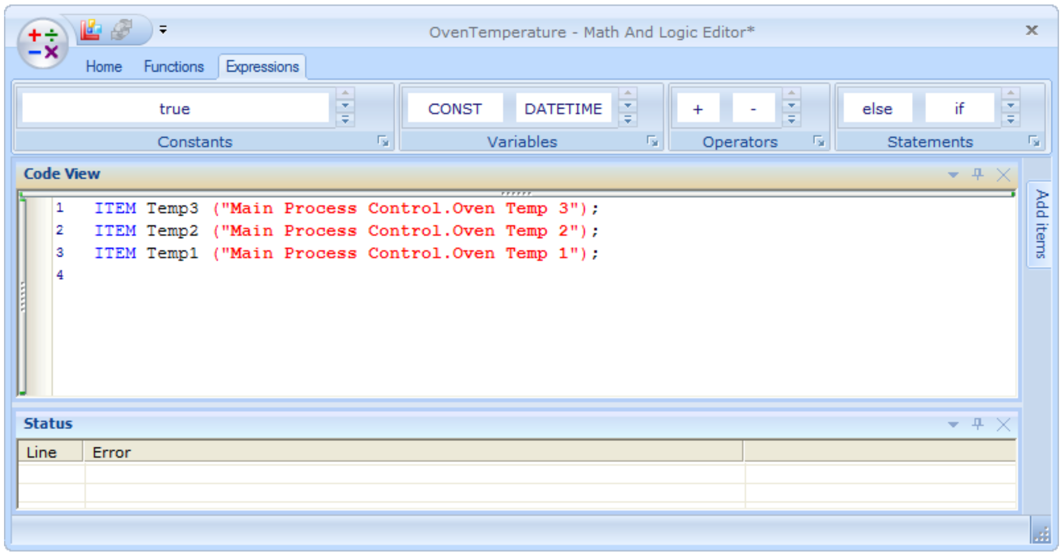
6. Click ***Add Item***.

   The item declaration will be added to the program.



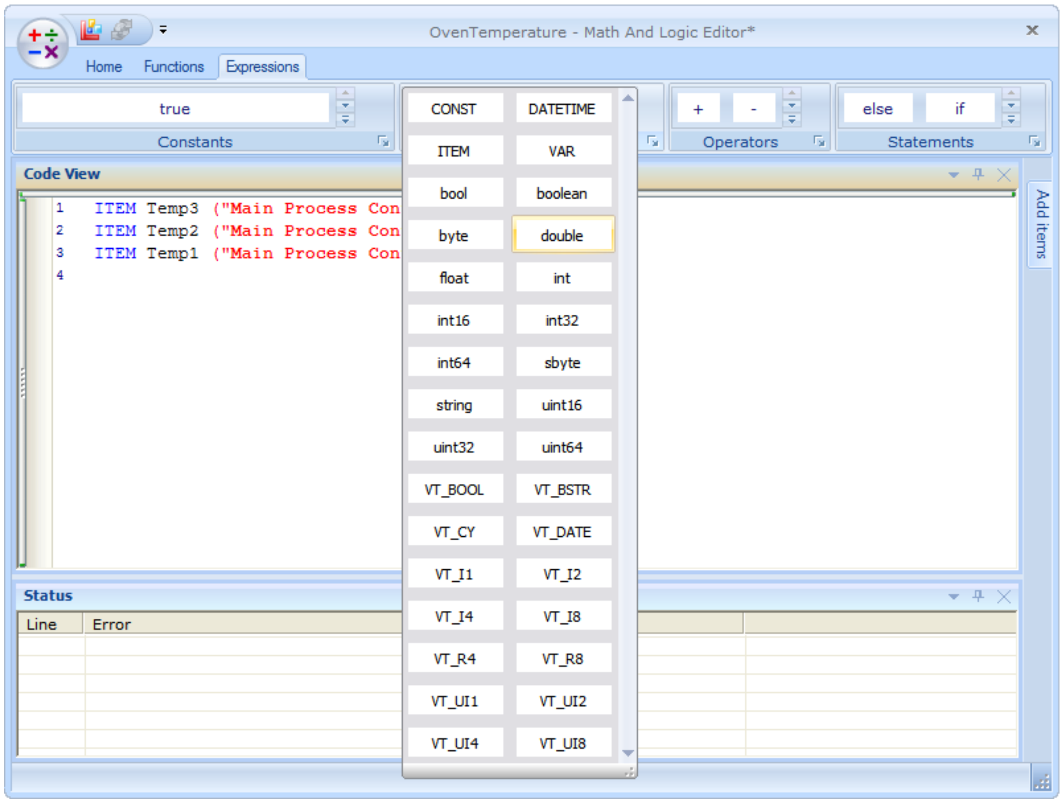
```
ITEM Temp1 ("Main Process Control.Oven Temp 1");
```

7. Repeat the preceding steps for the other data items, calling them ***Temp2*** and ***Temp3***.



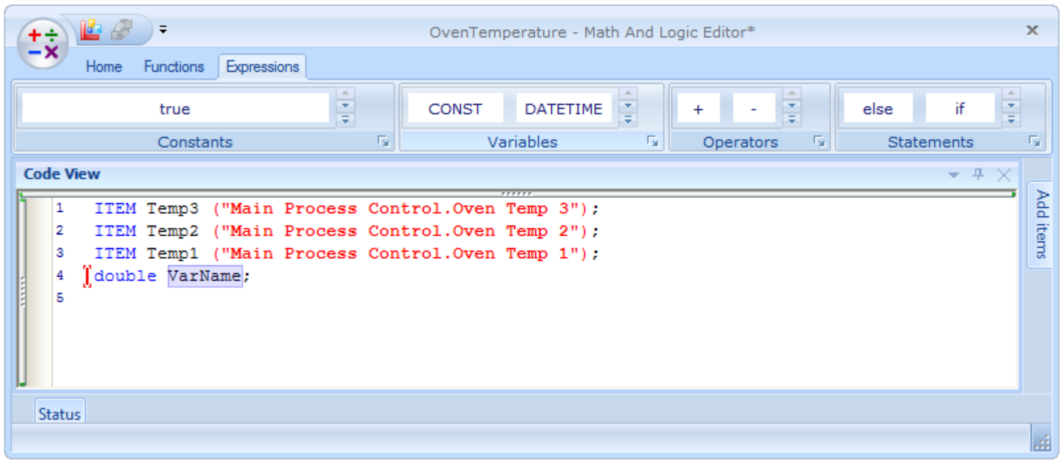
```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
```

The additional items appear in reverse order, because the editor always inserts new items at the top of the program.



```
ITEM Temp3 ("Main Process Con
ITEM Temp2 ("Main Process Con
ITEM Temp1 ("Main Process Con
```

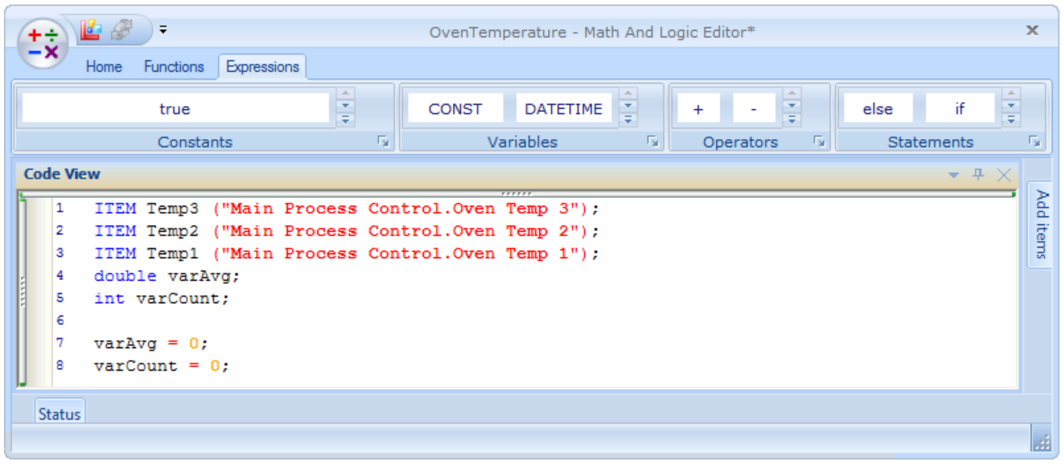
8. From the ***Variables*** group, select ***double***.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double VarName;
```

9. The editor will begin a VAR declaration statement. Complete it by typing ***varAvg;*** in the editing window.

10. Repeat the process to enter the declaration ***int varCount;*** on the next line.

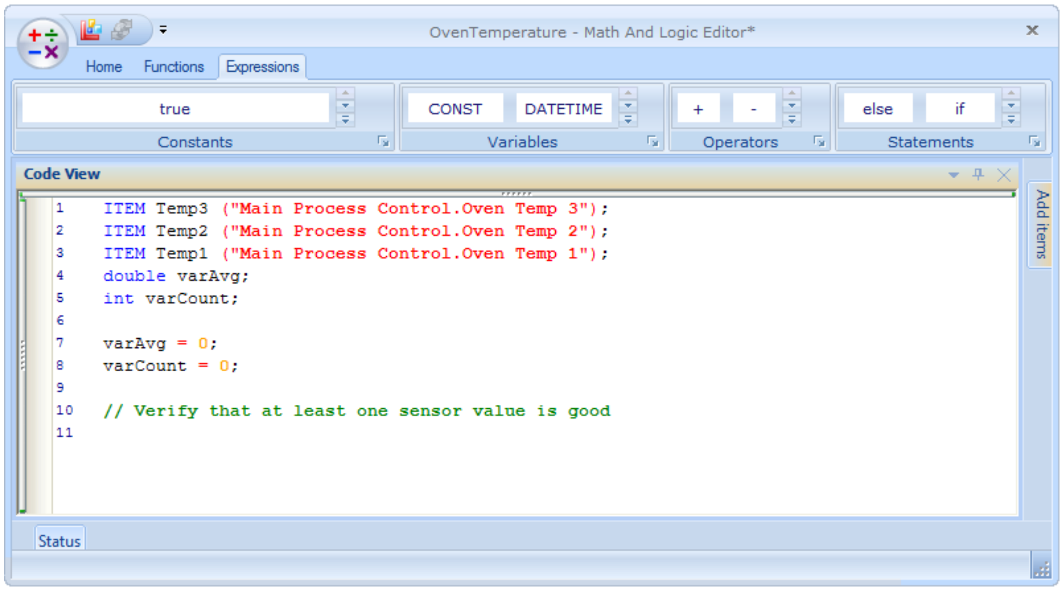


```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;
```

11. Add two lines with the following assignment statements:

    varAvg = 0;
    varCount = 0;

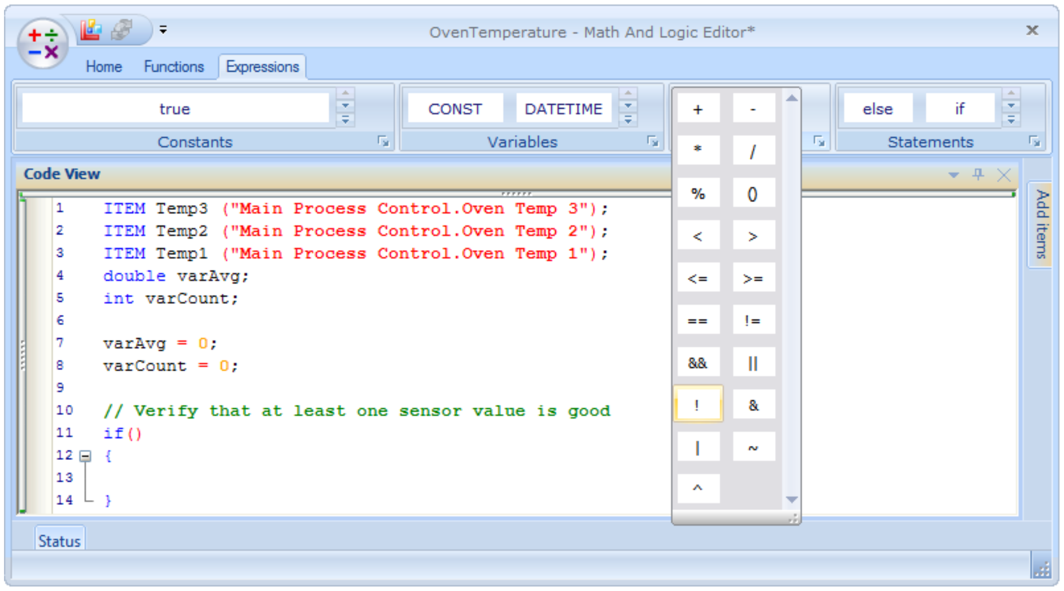
    These lines will initialize the two variables each time the program runs.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good

```

12. Type ***//*** to add a comment to your code. The double-slash and anything that follows it on the same line is considered to be a comment. The compiler will ignore it.

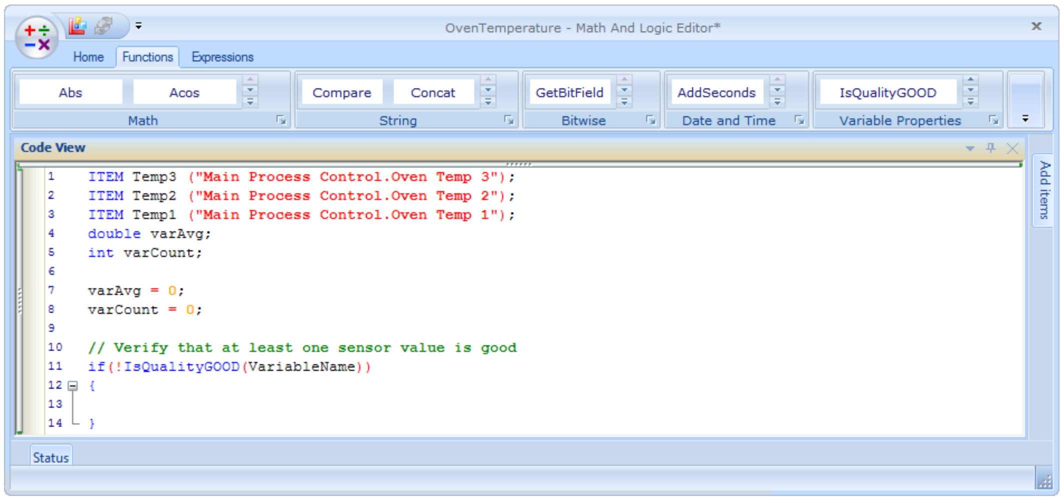
13. In the ***Statements*** group, click ***if*** to insert an if() structure.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if()
{

}
```

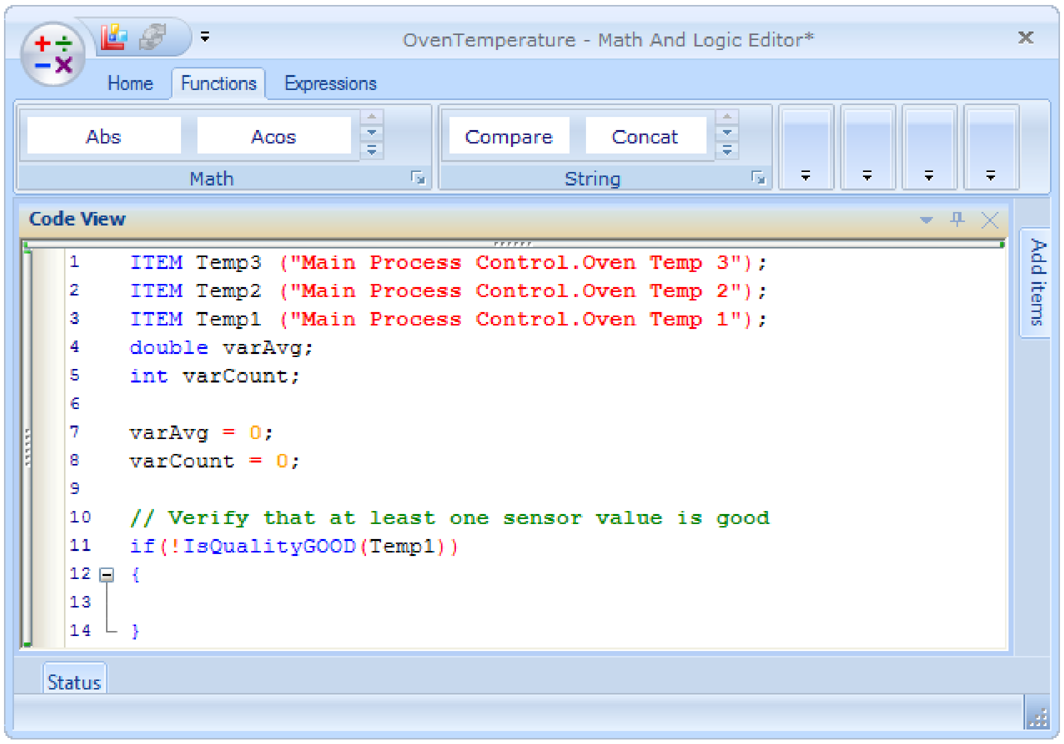
14. Place the cursor within the if() parentheses, then open the ***Operators*** menu and select ***!***, which is the logical NOT operator.

Note Click the expand button at the lower right corner of any of the groups to pop up a list of items available in that group, along with an explanation of their meaning and a sample syntax.



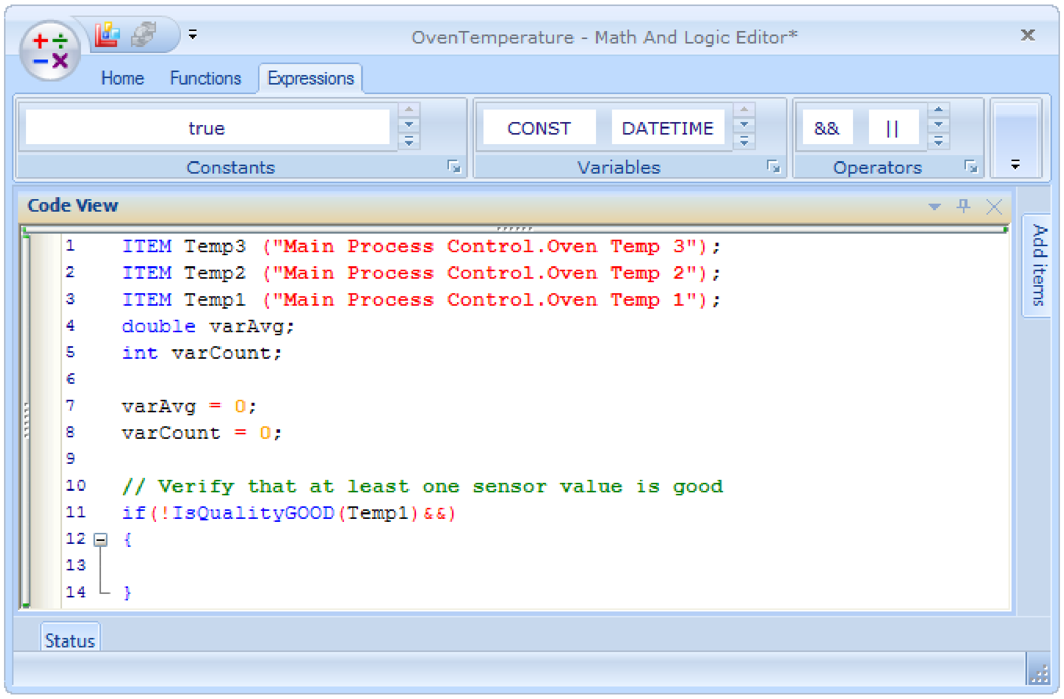
```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(VariableName))
{

}
```

15. Go to the ***Functions*** ribbon and select ***IsQualityGOOD*** from the ***OPC Quality*** group.



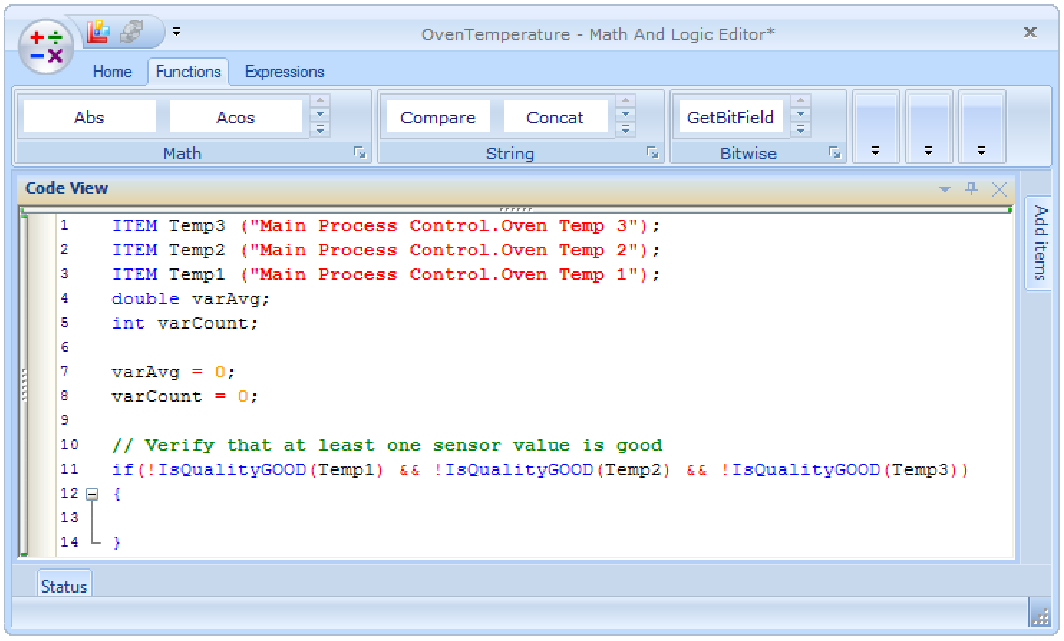
```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1))
{

}
```

16. The default argument for the function is VariableName. This is a hint to remind you to type the name of an OPC variable in its place. In this case, type ***Temp1***.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1)&&)
{

}
```

17. Use the ***Operators*** menu to enter the logical AND operator, ***&&*** into the expression.

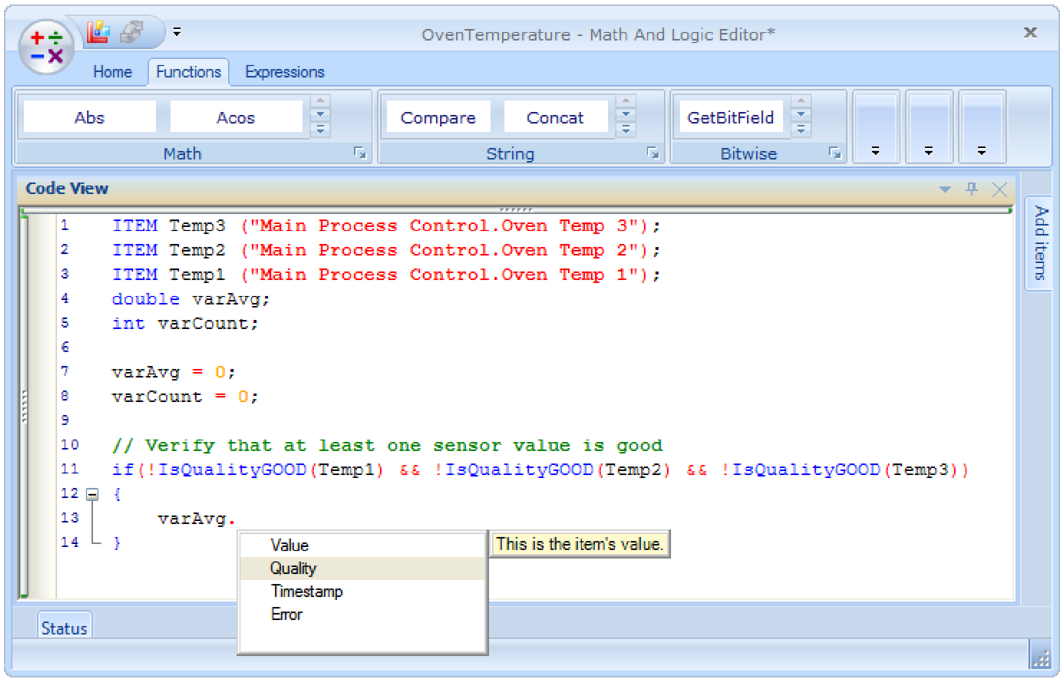


```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{

}
```

18. Repeat the preceding steps to complete the expression:

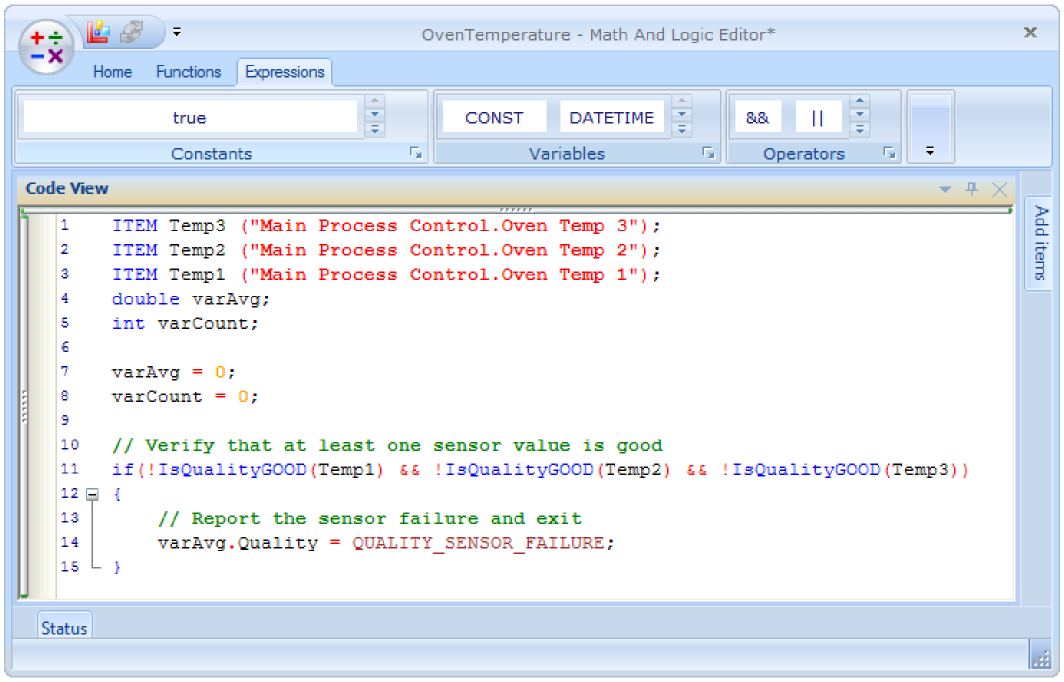
```
!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) &&
!IsQualityGOOD(Temp3)
```

Note The ribbon tools will help you to enter the expressions quickly and accurately. As you become more familiar with the editor, you may prefer to simply type the functions and operators directly.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{
    varAvg.
}
```

19. Place the cursor between the braces and type ***varAvg.*** (being sure to include the period). You will see a context menu appear, displaying the available properties of the object. Select ***Quality***.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{
    // Report the sensor failure and exit
    varAvg.Quality = QUALITY_SENSOR_FAILURE;
}
```

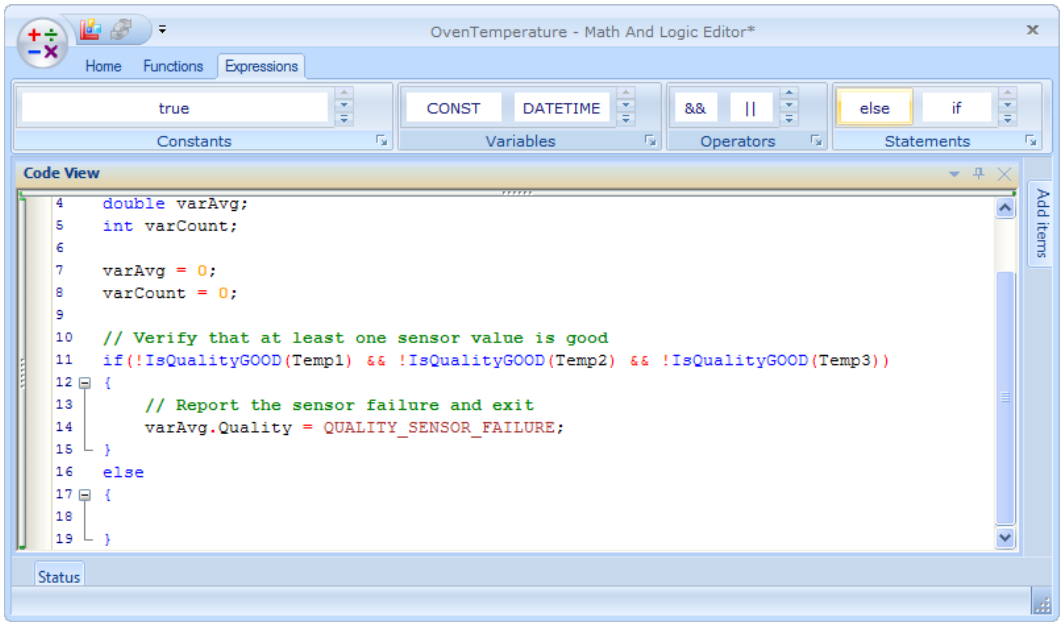
20. Type the assignment operator, ***=***, then go to the ***Expressions*** ribbon and select ***QUALITY_SENSOR_FAILURE*** from the ***Constants*** group.

Note If a variable that is returned from a program has a BAD quality, the value previously assigned to this variable becomes irrelevant.

21. Type a semicolon to complete the statement, then insert a comment line above the statement.

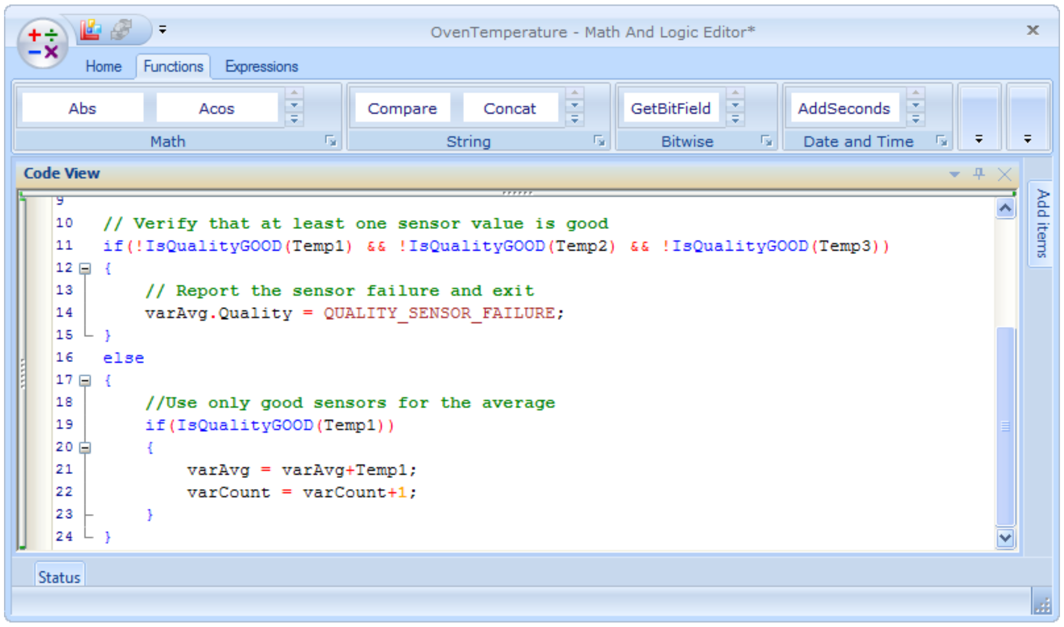
    The logic up to this point checks to see if none of the three sensors have good quality data, and, if so, reports a sensor failure. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

    Next, we will calculate the average of the good sensor readings.



```
double varAvg;
int varCount;

varAvg = 0;
varCount = 0;

// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{
    // Report the sensor failure and exit
    varAvg.Quality = QUALITY_SENSOR_FAILURE;
}
else
{

}
```

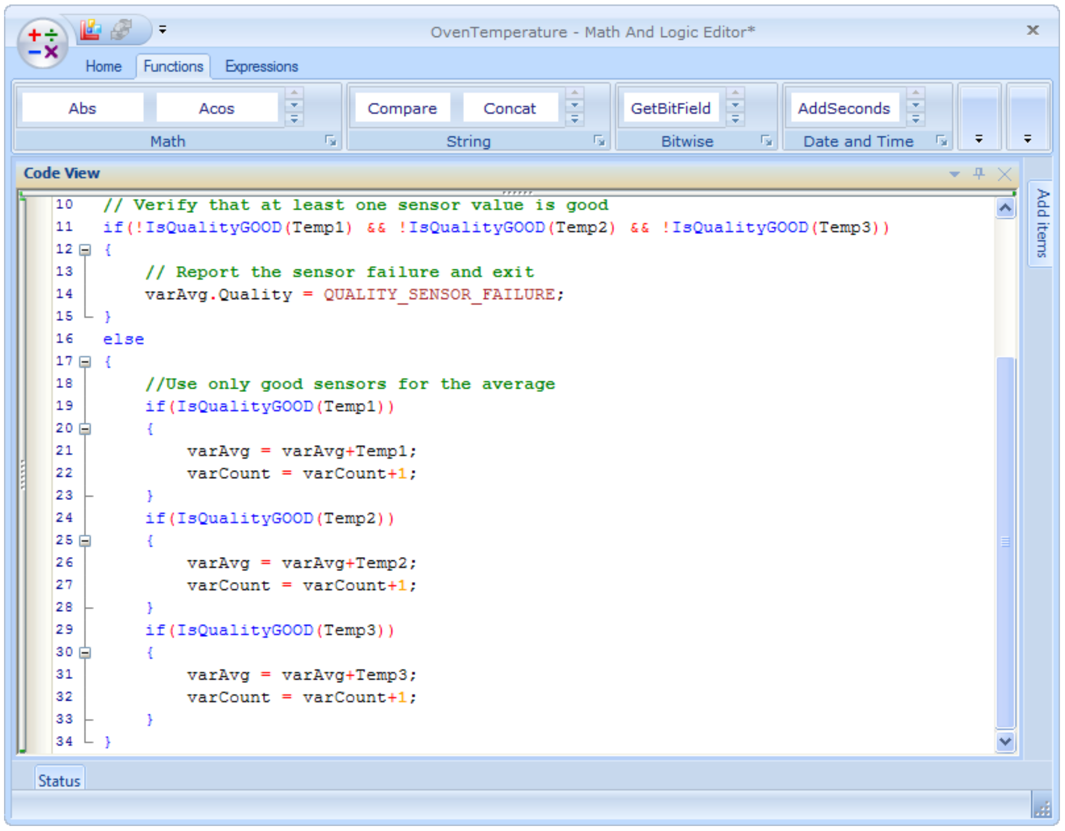
22. In the ***Statements*** group, select ***else***.



```
// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{
    // Report the sensor failure and exit
    varAvg.Quality = QUALITY_SENSOR_FAILURE;
}
else
{
    //Use only good sensors for the average
    if(IsQualityGOOD(Temp1))
    {
        varAvg = varAvg+Temp1;
        varCount = varCount+1;
    }
}
```

23. Use the techniques already discussed to enter the statements:

```
// Use only good sensors for the average
if(IsQualityGOOD(Temp1))
{
   varAvg = varAvg+Temp1;
   varCount = varCount+1;
}
```

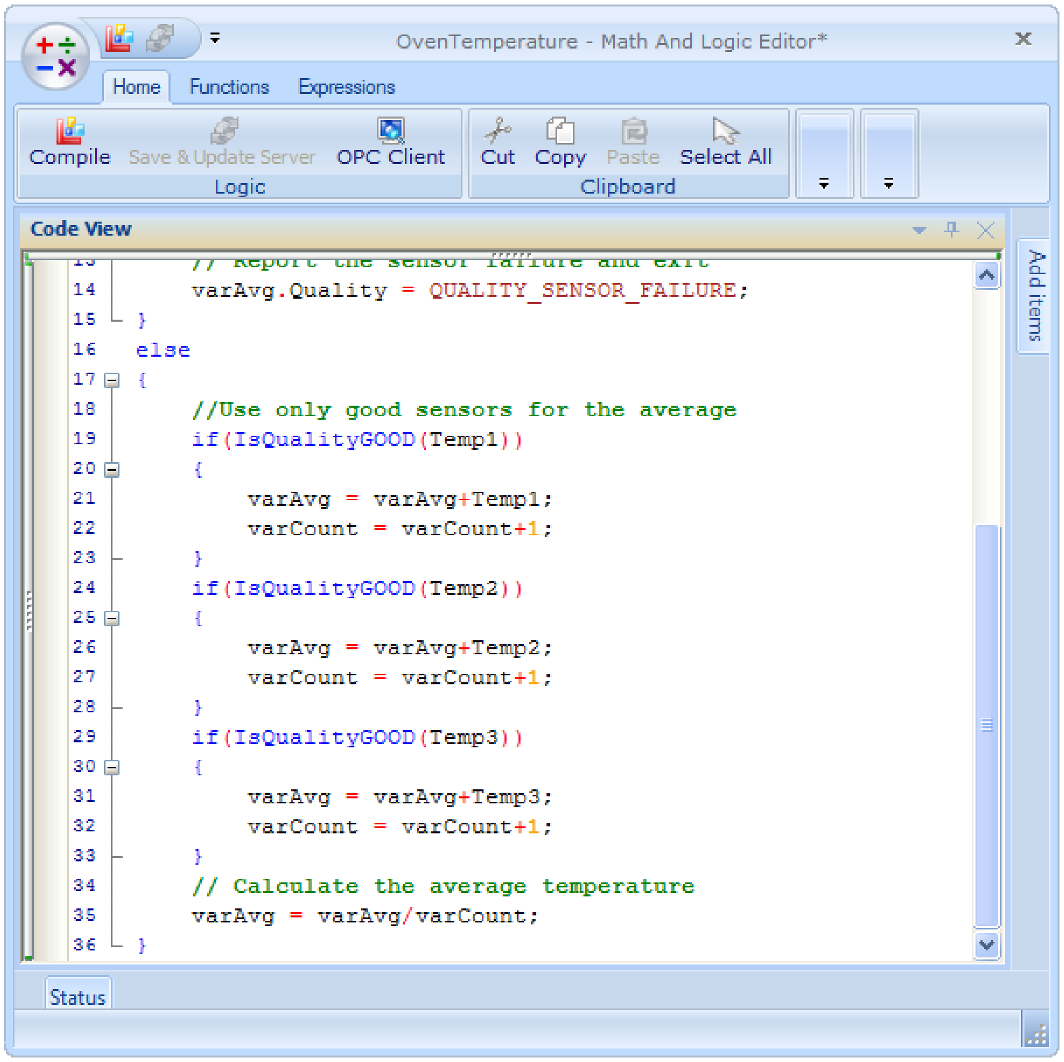


```
// Verify that at least one sensor value is good
if(!IsQualityGOOD(Temp1) && !IsQualityGOOD(Temp2) && !IsQualityGOOD(Temp3))
{
    // Report the sensor failure and exit
    varAvg.Quality = QUALITY_SENSOR_FAILURE;
}
else
{
    //Use only good sensors for the average
    if(IsQualityGOOD(Temp1))
    {
        varAvg = varAvg+Temp1;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp2))
    {
        varAvg = varAvg+Temp2;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp3))
    {
        varAvg = varAvg+Temp3;
        varCount = varCount+1;
    }
}
```

24. Go to the ***Home*** ribbon's ***Clipboard*** group. Use the ***Copy*** and ***Paste*** tools to make two additional copies of the preceding if statement.

25. Modify the copies to refer to ***Temp2*** and ***Temp3***, respectively.

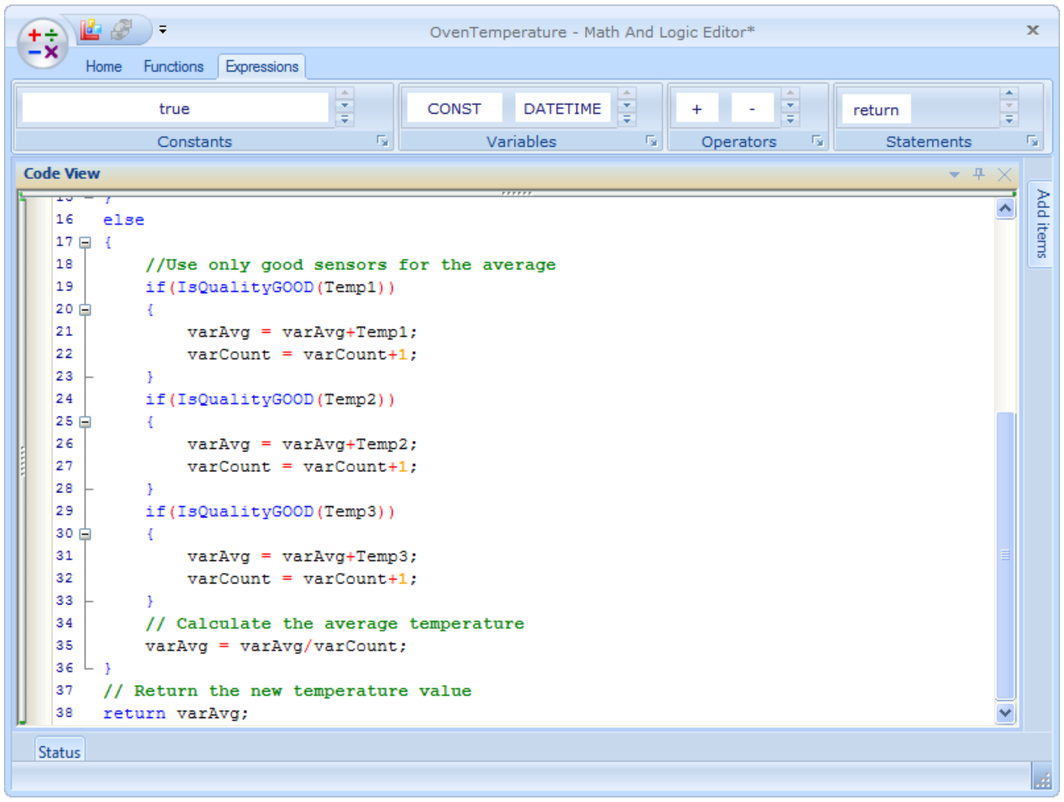
    Now we have the sum of the values from the good sensors. Next, we will calculate the average.



```
        varAvg.Quality = QUALITY_SENSOR_FAILURE;
    }
    else
    {
        //Use only good sensors for the average
        if(IsQualityGOOD(Temp1))
        {
            varAvg = varAvg+Temp1;
            varCount = varCount+1;
        }
        if(IsQualityGOOD(Temp2))
        {
            varAvg = varAvg+Temp2;
            varCount = varCount+1;
        }
        if(IsQualityGOOD(Temp3))
        {
            varAvg = varAvg+Temp3;
            varCount = varCount+1;
        }
        // Calculate the average temperature
        varAvg = varAvg/varCount;
    }
```

26. Within the scope of the else, type the expression:

```
varAvg = varAvg/varCount;
```

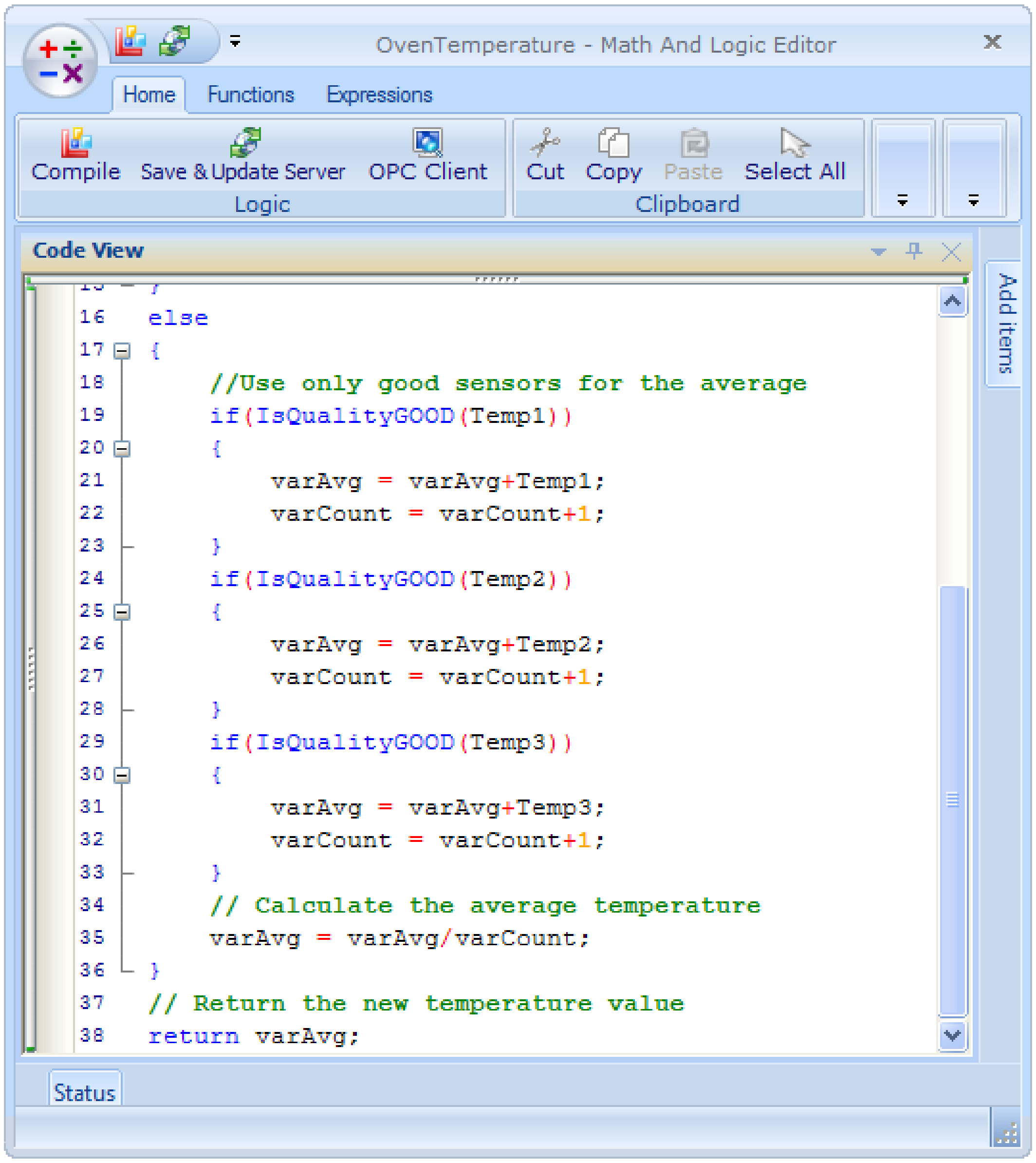


```
else
{
    //Use only good sensors for the average
    if(IsQualityGOOD(Temp1))
    {
        varAvg = varAvg+Temp1;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp2))
    {
        varAvg = varAvg+Temp2;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp3))
    {
        varAvg = varAvg+Temp3;
        varCount = varCount+1;
    }
    // Calculate the average temperature
    varAvg = varAvg/varCount;
}
// Return the new temperature value
return varAvg;
```

27. Go to the ***Expressions*** ribbon, ***Statements*** group, and select ***return***.

28. Type ***varAvg*** as the value to be returned.

    This completes the program editing. The last step is to compile the program.



```
else
{
    //Use only good sensors for the average
    if(IsQualityGOOD(Temp1))
    {
        varAvg = varAvg+Temp1;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp2))
    {
        varAvg = varAvg+Temp2;
        varCount = varCount+1;
    }
    if(IsQualityGOOD(Temp3))
    {
        varAvg = varAvg+Temp3;
        varCount = varCount+1;
    }
    // Calculate the average temperature
    varAvg = varAvg/varCount;
}
// Return the new temperature value
return varAvg;
```

29. On the ***Home*** ribbon, go to the ***Logic*** group.

30. Click the ***Compile*** button.

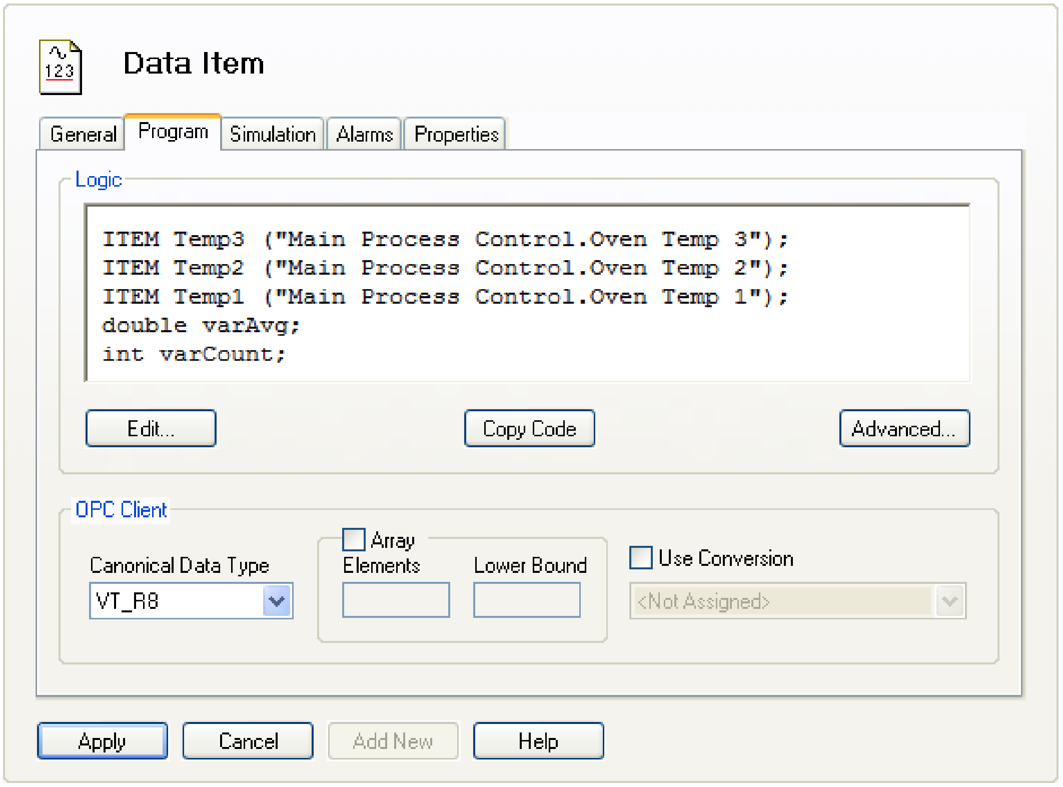
    The editor will save the program, and then compile it into the binary form that is executed by the server. If the compiler finds any errors, it will display them in the Status pane.

31. Click the ***Save & Update Server*** button to save the modified configuration and load it to the OPC server for execution.

Note It is not mandatory to perform the Save & Update Server operation within the Math & Logic editor. If you prefer, you can close the editor and do additional edits to your server configuration, and then Save & Update Server from the main menu or toolbar.

32. Close the Math & Logic Editor.

    You will return to the OPC Server Configuration Editor.



```
ITEM Temp3 ("Main Process Control.Oven Temp 3");
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1");
double varAvg;
int varCount;
```

Canonical Data Type: VT_R8

Note	If you make changes to the program, and leave the Math & Logic editor without compiling it, the program in the Logic window will be shown on a red background. The data item icon in the address space tree will also turn red. This is to remind you, that there is no executable code associated with your program.

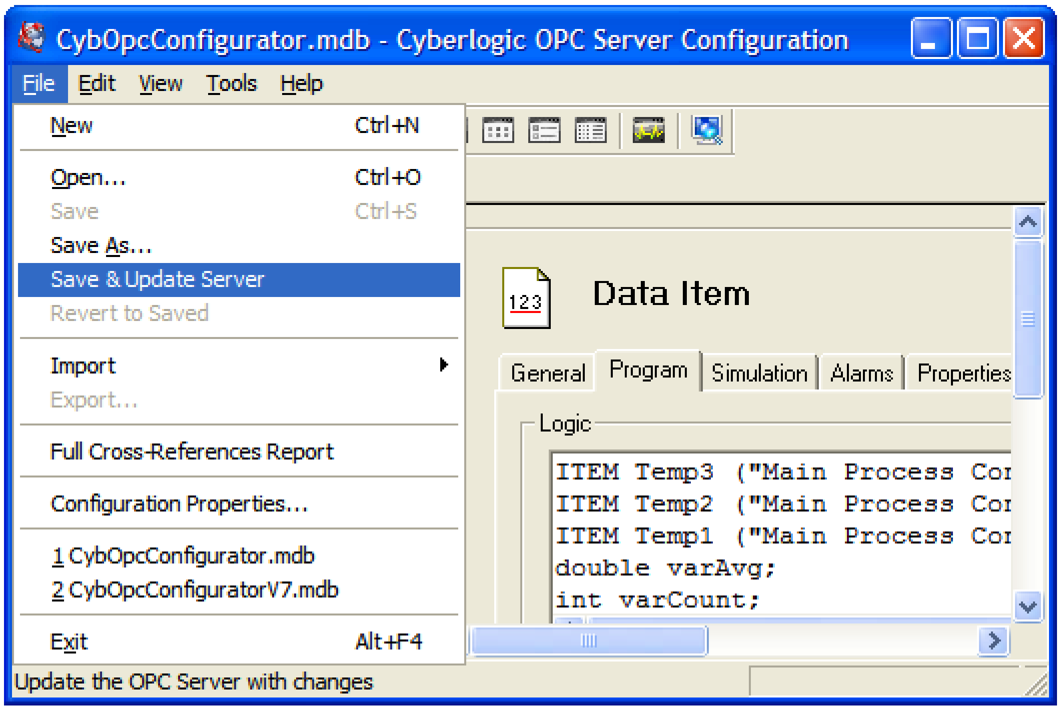
33. If you did not Save & Update Server within the Math & Logic editor, click ***Apply*** to save the program as part of the data item.

To continue, go to Saving the Configuration and Updating the Server.

Saving the Configuration and Updating the Server

If you did not Save & Update Server within the Math & Logic editor, you must do so before the server will be able to execute the program.

Caution! After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.


CybOpcConfigurator.mdb - Cyberlogic OPC Server Configuration
File Edit View Tools Help
New Ctrl+N
Open... Ctrl+O
Save Ctrl+S
Save As...
Save & Update Server
Revert to Saved
Import
Export...
Full Cross-References Report
Configuration Properties...
1 CybOpcConfigurator.mdb
2 CybOpcConfiguratorV7.mdb
Exit Alt+F4
Data Item
General
Program
Simulation
Alarms
Properties
Logic
Update the OPC Server with changes

```
ITEM Temp3 ("Main Process Co
ITEM Temp2 ("Main Process Co
ITEM Temp1 ("Main Process Co
double varAvg;
int varCount;
```

1. Open the ***File*** menu and select ***Save & Update Server***.

2. Be sure to repeat this step every time you change the configuration.

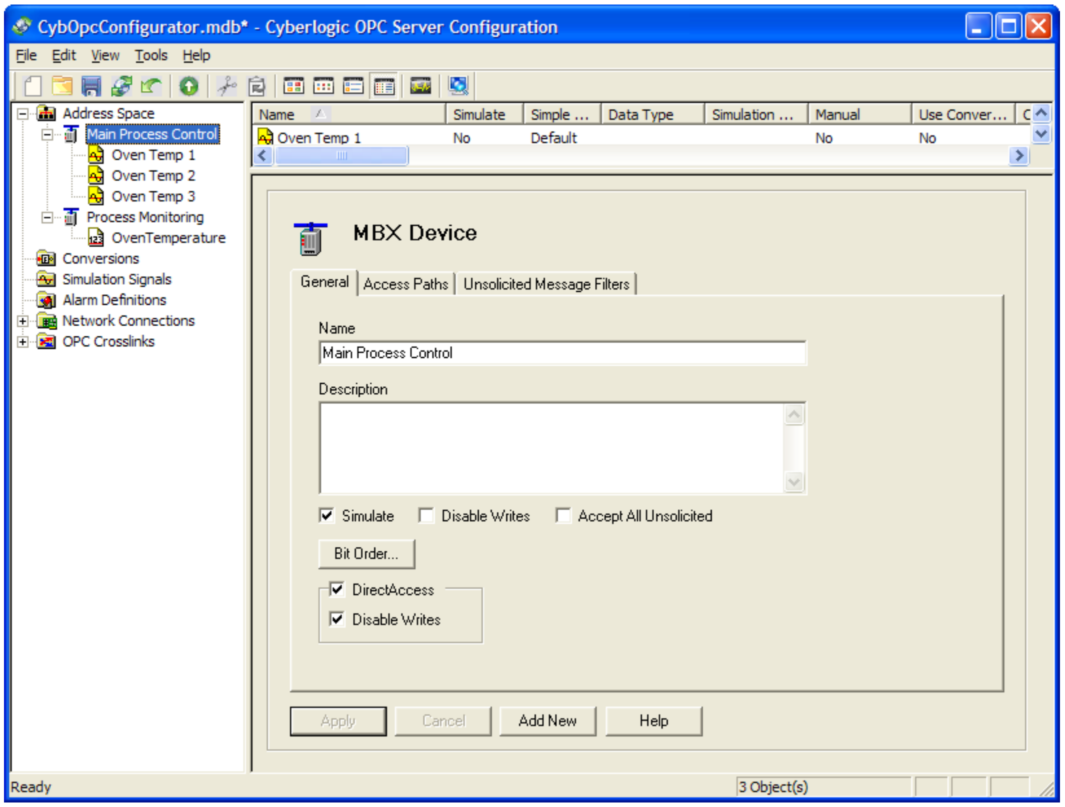
Your Math & Logic configuration is complete.

The next step, Verifying Your Configuration, will introduce you to the diagnostic features of the product.

Verifying Your Configuration

The Cyberlogic OPC Server Configuration Editor includes a built-in utility called Data Monitor. This diagnostic tool allows you to view the status and values for data items in the currently selected folder.

To test the Math & Logic program, we will simulate the three temperature inputs and use Data Monitor to set their values. We will then check the value of the Math & Logic data item to verify that it is correct.



Name	Simulate	Simple ...	Data Type	Simulation ...	Manual	Use Conver...
Oven Temp 1	No	Default			No	No

Name: Main Process Control

Description:

[✓] Simulate [ ] Disable Writes [ ] Accept All Unsolicited

[✓] DirectAccess

[✓] Disable Writes

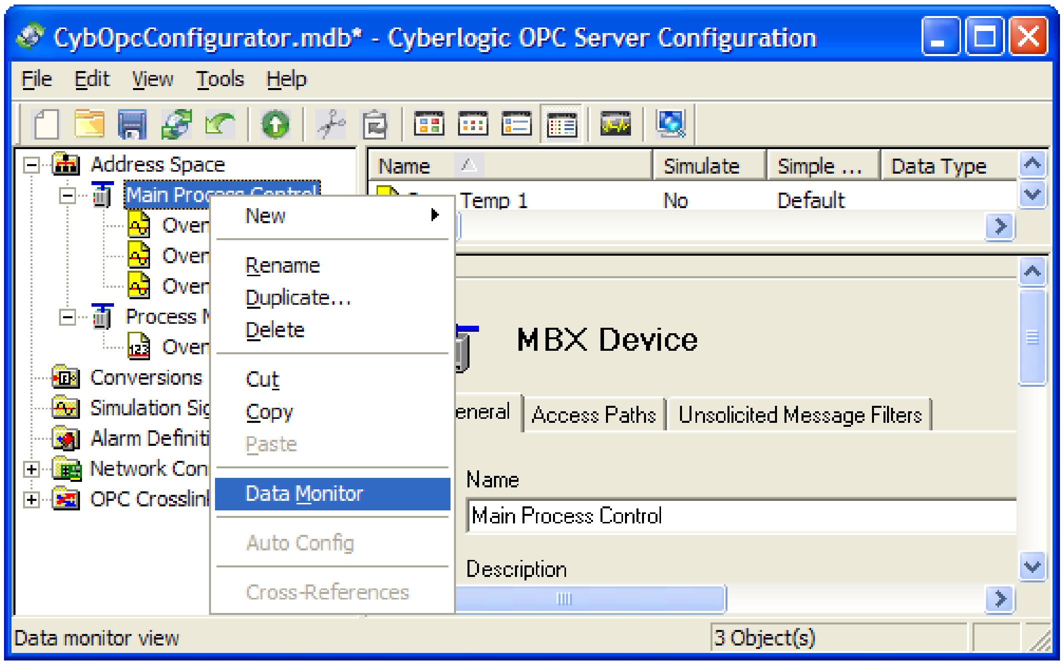
1. Select the device that contains the temperature sensors. In the example shown, it is called ***Main Process Control***.

2. On the ***General*** tab, check ***Simulate***.

3. Click ***Apply***.

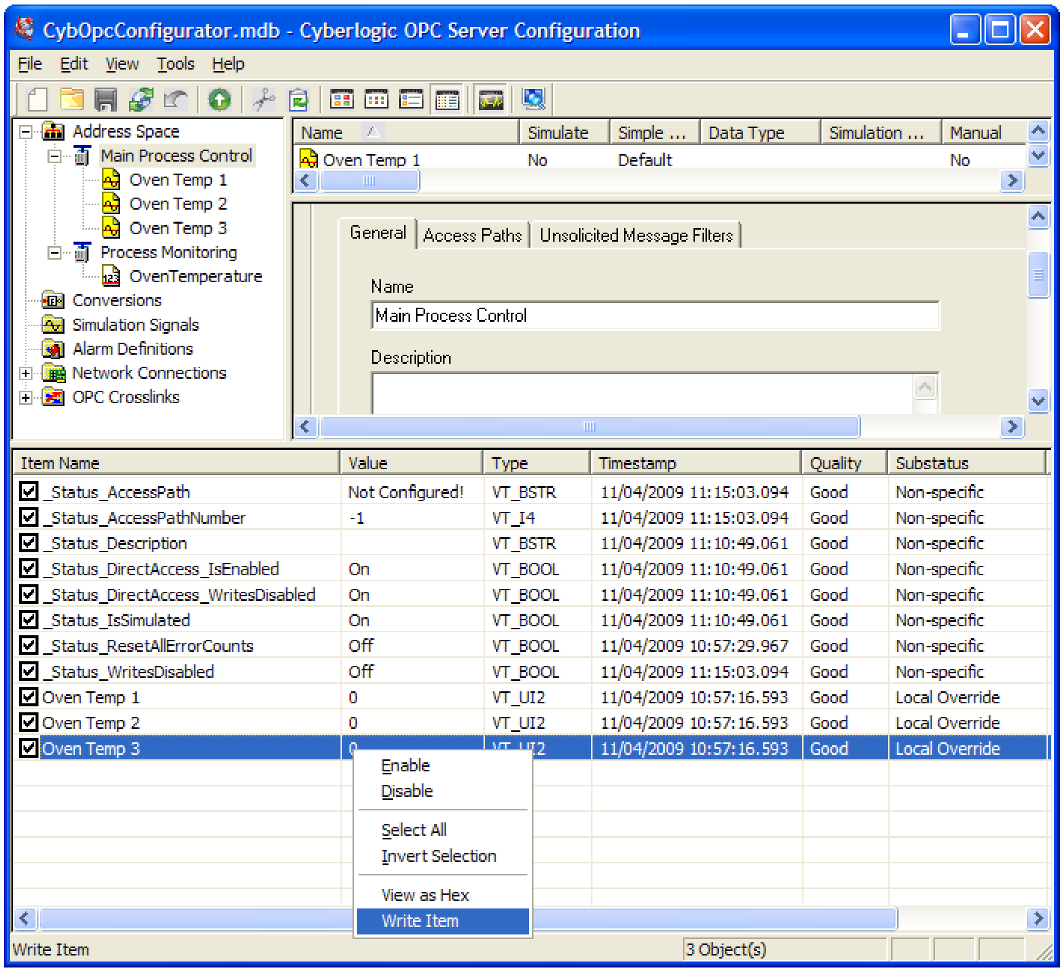
   The icons for the temperature sensors will turn yellow to indicate that their values are simulated.

4. On the toolbar, click the button to save and update the server.



5. Right-click on the ***Main Process Control*** device and select ***Data Monitor*** from the context menu.

   The Data Monitor pane will open to show the values of the sensors, along with status information.



Name	Simulate	Simple ...	Data Type	Simulation ...	Manual
Oven Temp 1	No	Default			No

Name: Main Process Control

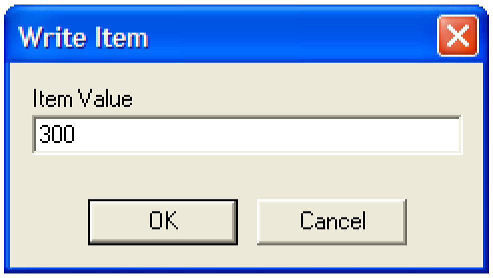
Description:

Item Name	Value	Type	Timestamp	Quality	Substatus
_Status_AccessPath	Not Configured!	VT_BSTR	11/04/2009 11:15:03.094	Good	Non-specific
_Status_AccessPathNumber	-1	VT_I4	11/04/2009 11:15:03.094	Good	Non-specific
_Status_Description		VT_BSTR	11/04/2009 11:10:49.061	Good	Non-specific
_Status_DirectAccess_IsEnabled	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_DirectAccess_WritesDisabled	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_IsSimulated	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_ResetAllErrorCounts	Off	VT_BOOL	11/04/2009 10:57:29.967	Good	Non-specific
_Status_WritesDisabled	Off	VT_BOOL	11/04/2009 11:15:03.094	Good	Non-specific
Oven Temp 1	0	VT_UI2	11/04/2009 10:57:16.593	Good	Local Override
Oven Temp 2	0	VT_UI2	11/04/2009 10:57:16.593	Good	Local Override
Oven Temp 3	0	VT_UI2	11/04/2009 10:57:16.593	Good	Local Override

Write Item

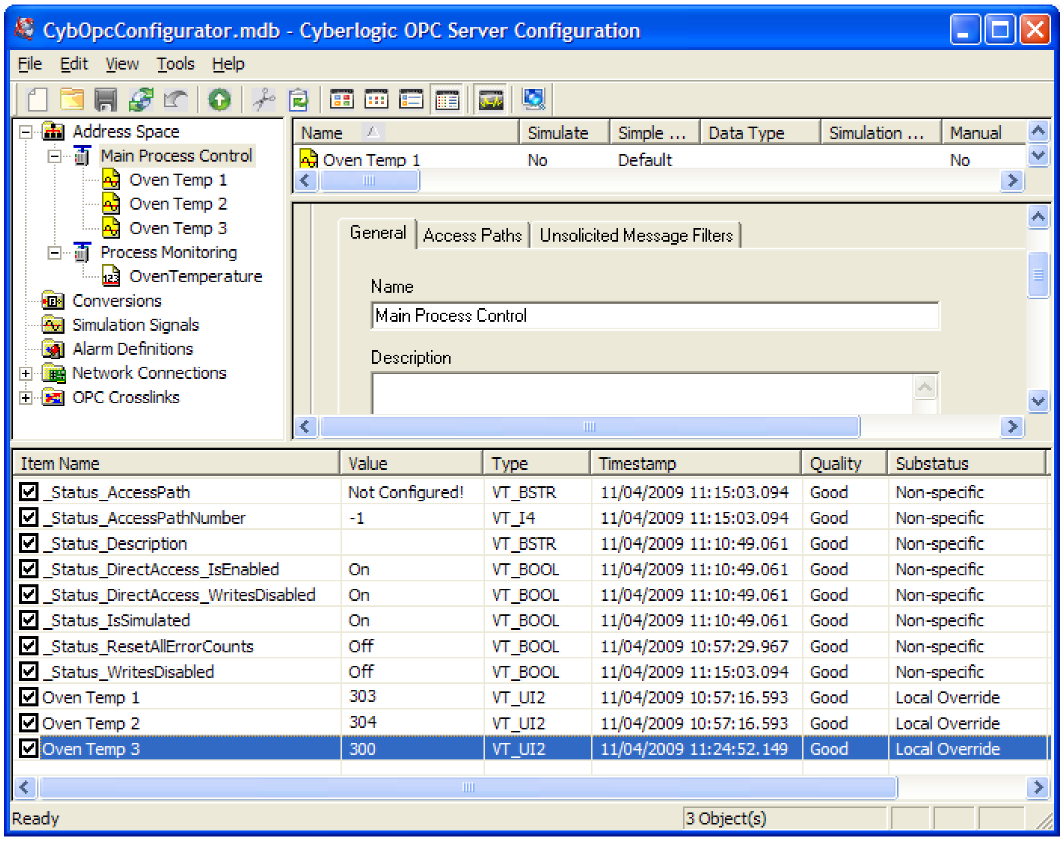
Notice that the Substatus for the three Oven Temp values is Local Override. This indicates that the values are simulated.

6. Right-click on ***Oven Temp 3*** and select ***Write Item*** from the context menu.



Item Value: 300

7. In the ***Write Item*** dialog box, enter a value for the item, and then click ***OK***.



Name	Simulate	Simple ...	Data Type	Simulation ...	Manual
Oven Temp 1	No	Default			No

Name: Main Process Control

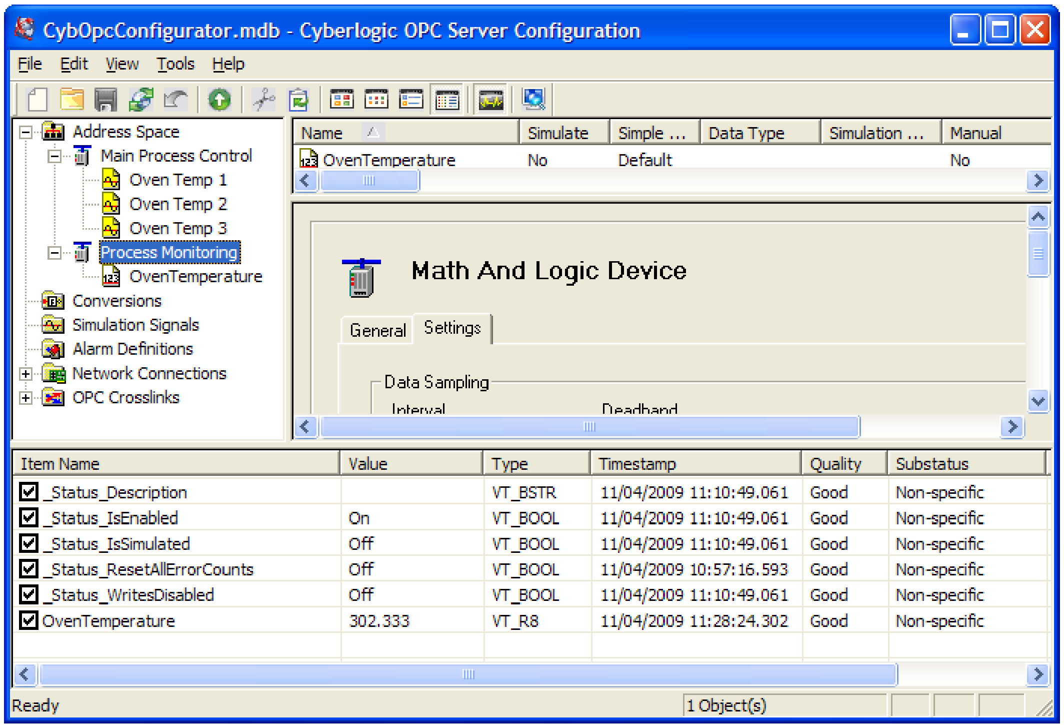
Description:

Item Name	Value	Type	Timestamp	Quality	Substatus
_Status_AccessPath	Not Configured!	VT_BSTR	11/04/2009 11:15:03.094	Good	Non-specific
_Status_AccessPathNumber	-1	VT_I4	11/04/2009 11:15:03.094	Good	Non-specific
_Status_Description		VT_BSTR	11/04/2009 11:10:49.061	Good	Non-specific
_Status_DirectAccess_IsEnabled	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_DirectAccess_WritesDisabled	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_IsSimulated	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_ResetAllErrorCounts	Off	VT_BOOL	11/04/2009 10:57:29.967	Good	Non-specific
_Status_WritesDisabled	Off	VT_BOOL	11/04/2009 11:15:03.094	Good	Non-specific
Oven Temp 1	303	VT_UI2	11/04/2009 10:57:16.593	Good	Local Override
Oven Temp 2	304	VT_UI2	11/04/2009 10:57:16.593	Good	Local Override
Oven Temp 3	300	VT_UI2	11/04/2009 11:24:52.149	Good	Local Override

8. Verify that the value was written as expected, then repeat the procedure to set values for the other two sensors.

9. Click on the ***Process Monitoring*** Math & Logic device.

   The Data Monitor pane will show the value of the OvenTemperature Math & Logic data item, along with status information.



Name	Simulate	Simple ...	Data Type	Simulation ...	Manual
OvenTemperature	No	Default			No

Item Name	Value	Type	Timestamp	Quality	Substatus
_Status_Description		VT_BSTR	11/04/2009 11:10:49.061	Good	Non-specific
_Status_IsEnabled	On	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_IsSimulated	Off	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
_Status_ResetAllErrorCounts	Off	VT_BOOL	11/04/2009 10:57:16.593	Good	Non-specific
_Status_WritesDisabled	Off	VT_BOOL	11/04/2009 11:10:49.061	Good	Non-specific
OvenTemperature	302.333	VT_R8	11/04/2009 11:28:24.302	Good	Non-specific

10. Verify that the value of ***OvenTemperature*** is the average of the three simulated temperature sensor values.

11. Select ***Main Process Control*** again and change the temperature sensor values. When you return to viewing ***Process Monitoring***, you should see a corresponding change in the ***OvenTemperature*** value.

This concludes the Quick-Start Guide. To learn more about the features of the server, refer to the Theory of Operation section. To learn more about configuration, refer to the Configuration Editor Reference. To learn more about verifying your configuration and troubleshooting tools, refer to the Cyberlogic OPC Server Help.

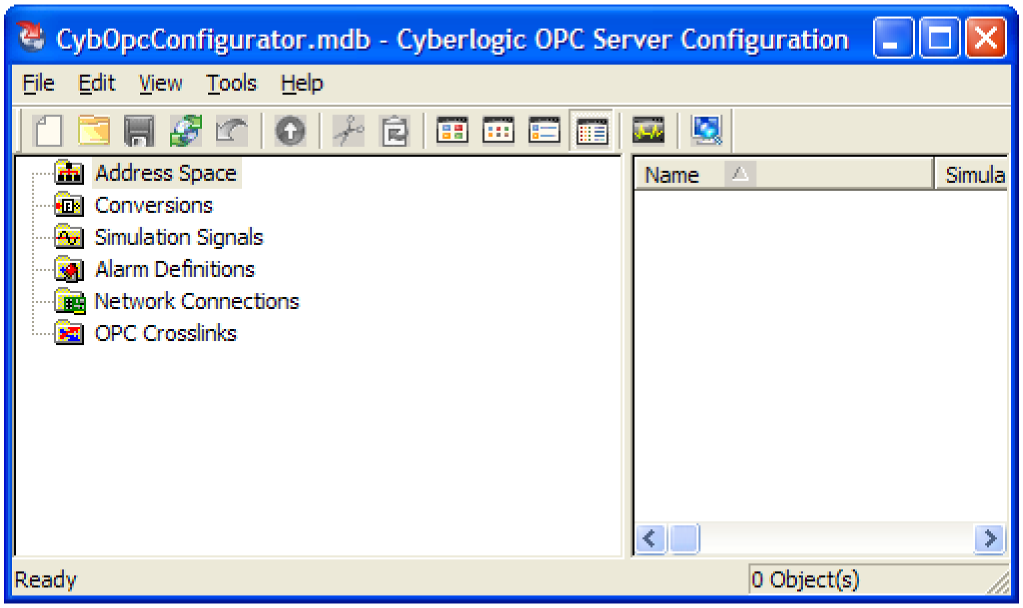
CONFIGURATION EDITOR REFERENCE

Before you can use the OPC server, you must configure it by using the OPC Server Configuration Editor. Math & Logic users must configure the Address Space tree. In addition, users who want to obtain data from PLCs or other OPC servers must configure the Network Connections tree. The remaining trees (Conversions, Simulation Signals, Alarm Definitions and OPC Crosslinks) are optional features used by some systems.

This section describes the editor features that you will use to configure Math & Logic. If you are a new user and want a procedure to guide you through a typical configuration session, refer to the Quick Start Guide.

Caution! After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.

To launch the editor from the Windows ***Start*** menu, go to ***Cyberlogic Suites***, then open the ***Configuration*** sub-menu, and then select ***OPC Server***.



The left pane of the main workspace window includes the six main configuration trees. The first part of this reference explains how to create and configure Math & Logic in the Address Space tree. This includes three sections:

- Math & Logic Devices
- Math & Logic Data Items
- Math & Logic Editor

Following that is a very brief description of the Conversions, Simulation Signals, Alarm Definitions, Network Connections and OPC Crosslinks trees. For a full discussion of these trees and other important topics including configuration import/export, editor options and connecting to OPC client software, please refer to the Cyberlogic OPC Server Help.

The last sections in this configuration editor reference cover important tips and techniques for Saving and Undoing Configuration Changes, Configuration Import/Export and Editor Options

Address Space

The Address Space Tree describes the hierarchical address structure of the Cyberlogic OPC Server. The branches of the tree that relate to Math & Logic are Device Folders, Math & Logic Devices, and Folders. Its "leaves" are Math & Logic Data Items. The intent of this structure is to permit the user to organize the data items into logical groups.

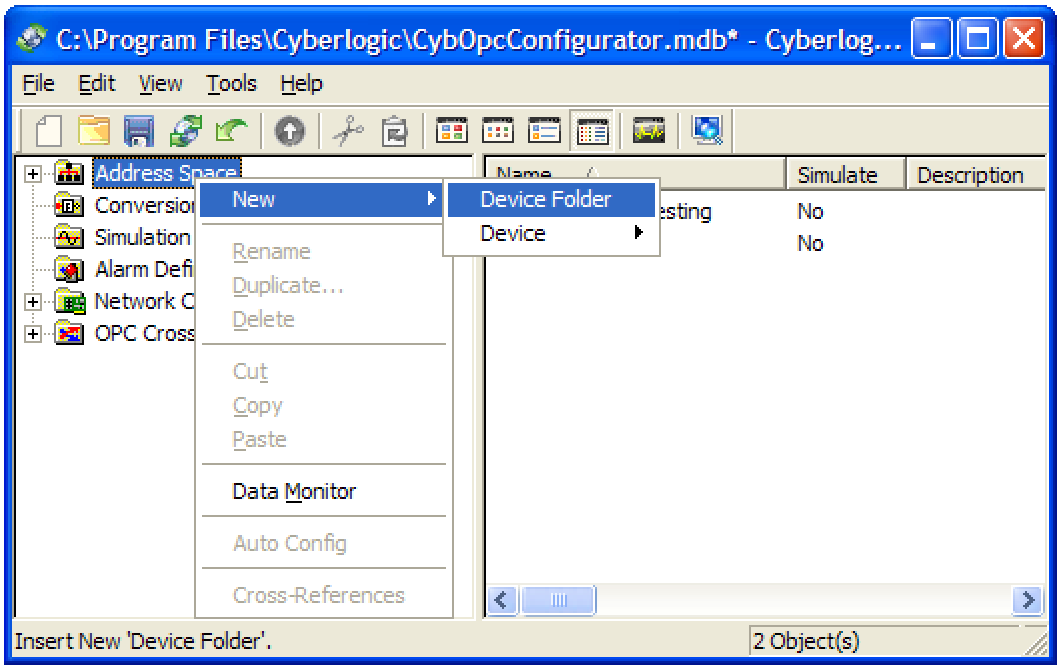
Device Folders

A device folder groups devices and other device folders. You can place a device folder directly under the Address Space root folder or under another device folder, up to four levels deep.

Caution!

After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.

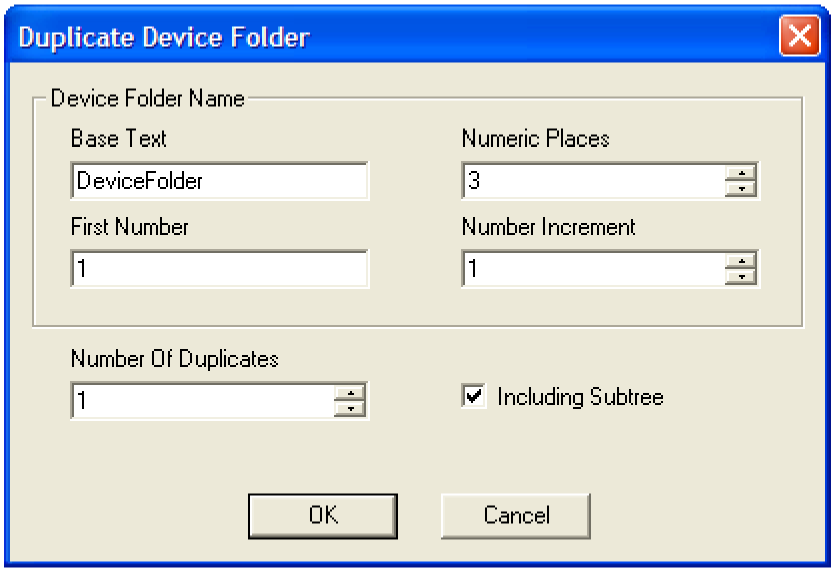
Creating a New Device Folder



Right-click on the Address Space root folder or an existing device folder. Select ***New*** and then ***Device Folder*** from the context menu.

Duplicating a Device Folder

You can create multiple device folders in a single operation by duplicating an existing one. This can help you to quickly create similarly-configured device folders. To duplicate a device folder, right-click on it and select ***Duplicate...*** from the context menu.


Duplicate Device Folder

Device Folder Name

Base Text: DeviceFolder

Numeric Places: 3

First Number: 1

Number Increment: 1

Number Of Duplicates: 1

[✓] Including Subtree

The above dialog box opens. You must specify how the duplicates are to be named by entering values for the ***Base Text***, ***First Number***, ***Numeric Places*** and ***Number Increment*** fields. To name the duplicated device folders, the editor begins with the base text and appends a number to it. The first duplicate uses the First Number value with the specified number of digits. The editor then adds Number Increment to that value for each of the remaining duplicates.

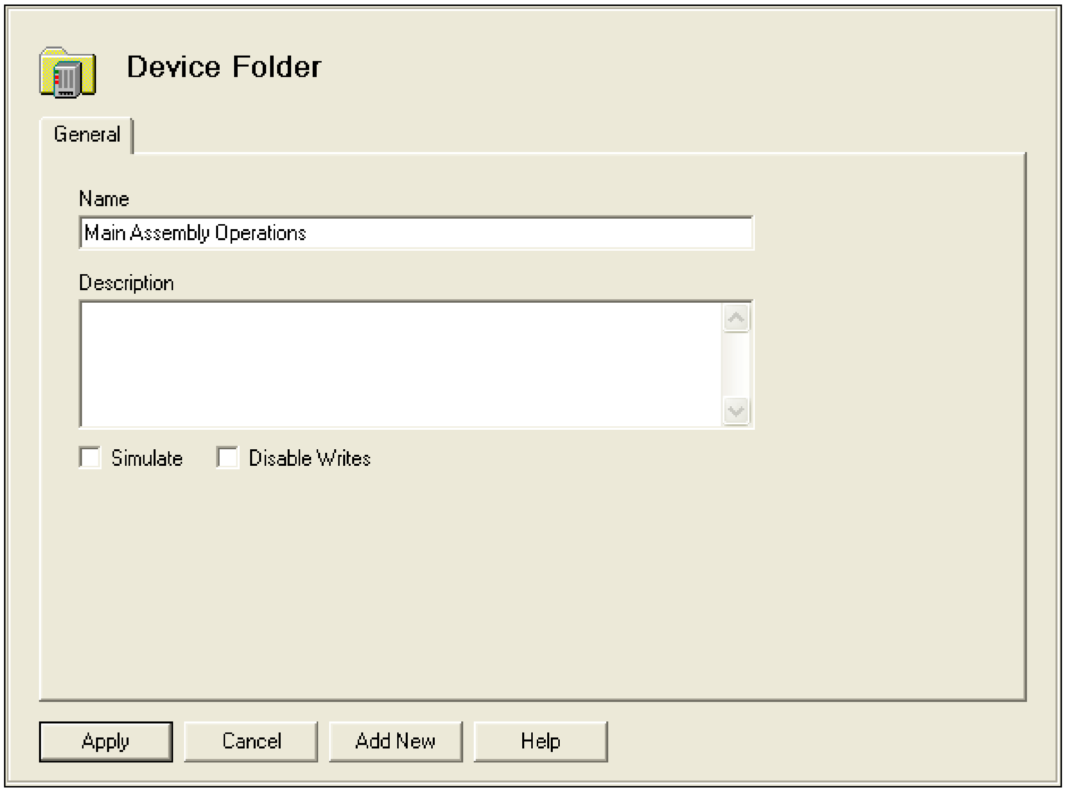
As an example, if Numeric Places is 3 and First Number is 2, the number 002 will be appended to the base text.

Use the ***Number Of Duplicates*** field to specify the number of device folders you wish to create. If you want to duplicate all branches within the original device folder, check the ***Including Subtree*** checkbox.

Deleting a Device Folder

To delete an existing device folder, select it and press the ***Delete*** key, or right-click on the device folder and select ***Delete*** from the context menu.

General Tab



Name: Main Assembly Operations

Description:

Simulate ☐ Disable Writes ☐

Name

The Name identifies this device folder. It can be up to 50 characters long, may contain spaces, but must not begin with a space. It also may not contain any periods.

Description

This optional field further describes the device folder. It can be up to 255 characters long.

Simulate

Check this box to enable data simulation for all data items at this level or below. This provides a quick way to switch between real and simulated data for a large number of data items. Refer to the Cyberlogic OPC Server Help for a full discussion about simulating data.

Note If the Simulate checkbox is grayed-out, it indicates that simulation has already been selected at a higher level.

Disable Writes

Check this box to disable write requests for all data items at this level or below. By default, this box is not checked and writes are enabled.

Note If the Disable Writes checkbox is grayed-out, it indicates that writes have already been disabled a higher level.

Math & Logic Devices

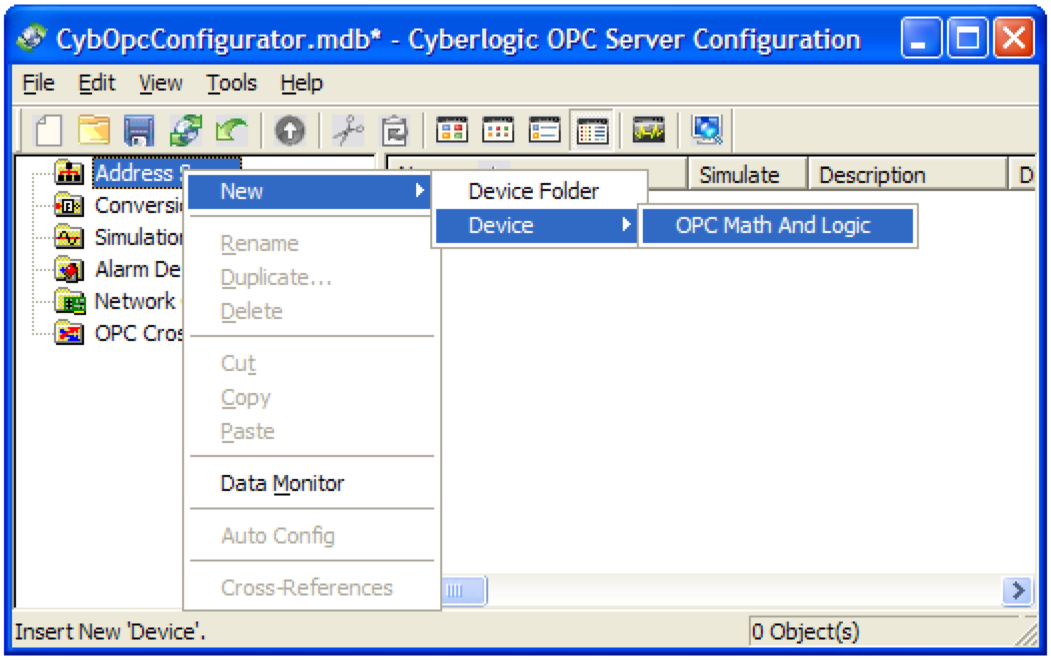
A Math & Logic device contains Math & Logic data items. The program enable and execution criteria are configured in the device, so users typically group data items that have the same criteria within the same device. You can place Math & Logic devices directly in the Address Space root folder or in a device folder. A Math & Logic device also functions as a folder; it can contain folders and Math & Logic data items.

Note All programs under a Math & Logic device execute in the context of a single thread. That means that if multiple programs are ready for execution, they will execute sequentially in the order in which they became ready for execution.

If your application requires concurrency, or if you want to better utilize a multi-core CPU, you should create multiple Math & Logic devices, and group the data items (programs) appropriately.

Caution! After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.

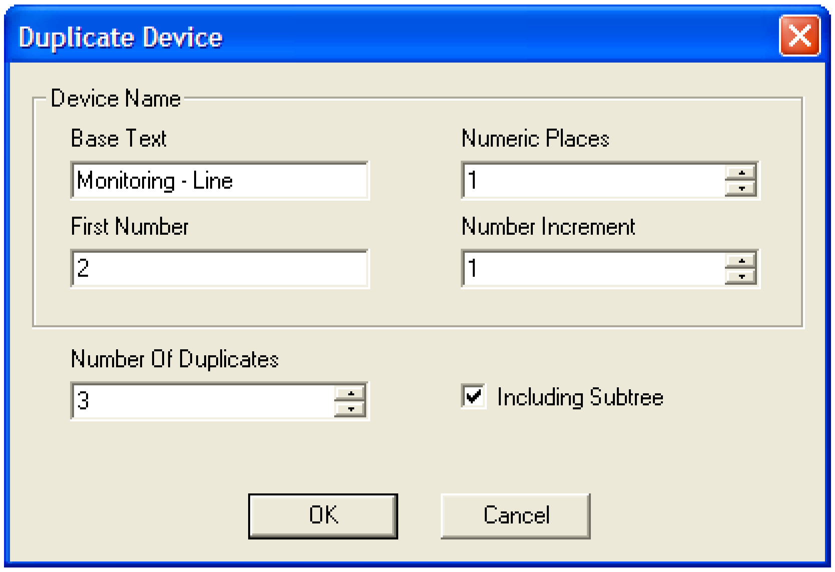
Creating a New Math & Logic Device



Right-click on the Address Space root folder or a device folder. Select ***New*** and then ***Device*** and then ***Math & Logic*** from the context menu.

Duplicating a Math & Logic Device

You can create multiple Math & Logic Devices in a single operation by duplicating an existing one. This can help you to quickly create similarly-configured devices. To duplicate a Math & Logic Device, right-click on it and select ***Duplicate...*** from the context menu.


Duplicate Device

Device Name

Base Text: Monitoring - Line

Numeric Places: 1

First Number: 2

Number Increment: 1

Number Of Duplicates: 3

[✓] Including Subtree

The above dialog box opens. You must specify how the duplicates are to be named by entering values for the ***Base Text***, ***First Number***, ***Numeric Places*** and ***Number Increment*** fields. To name the duplicates, the editor begins with the base text and appends a number to it. The first duplicate uses the selected First Number value with the specified number of digits. The editor then adds Number Increment to that value for each of the remaining duplicates.

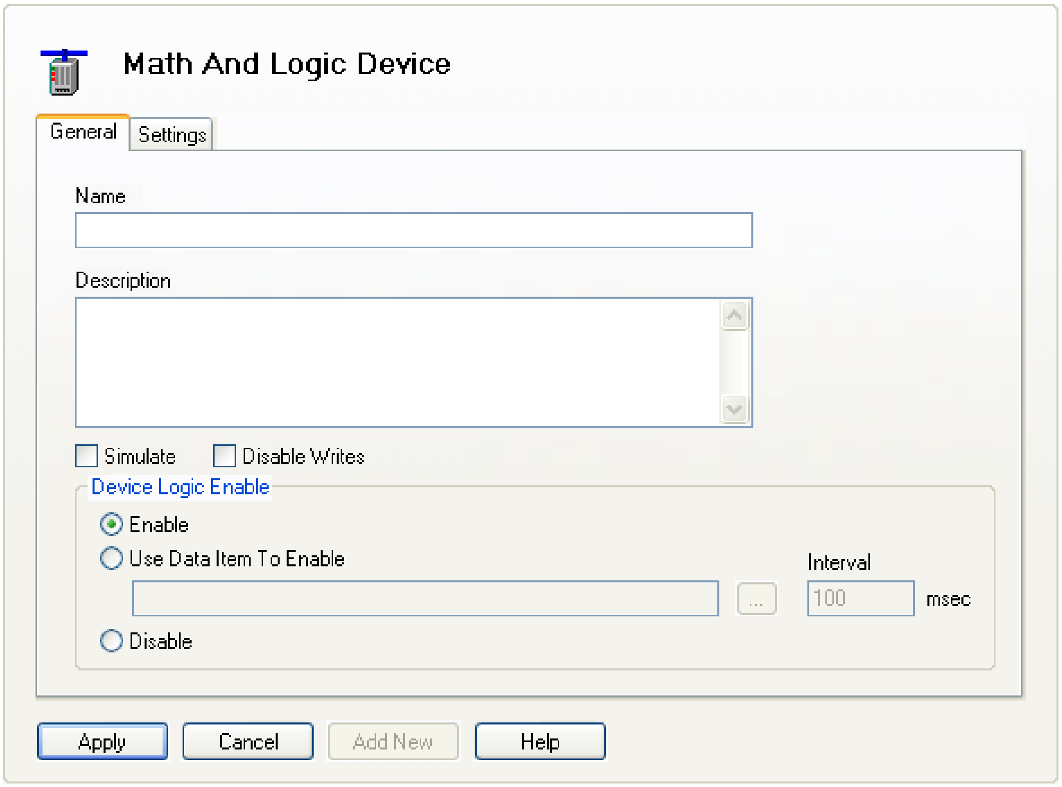
Use the ***Number Of Duplicates*** field to specify the number of Math & Logic Devices you wish to create. If you want to duplicate all branches within the original device, check the ***Including Subtree*** checkbox.

In the example, a single digit will be appended to the base text *Monitoring – Line* (there is a space at the end). The numbering begins with 2, and three duplicates will be created. They will be called *Monitoring – Line 2, Monitoring – Line 3* and *Monitoring – Line 4*.

Deleting a Math & Logic Device

To delete an existing Math & Logic device, select it and press the ***Delete*** key, or right-click on the device and select ***Delete*** from the context menu.

General Tab


Math And Logic Device

General | Settings

Name:

Description:

☐ Simulate ☐ Disable Writes

Device Logic Enable

[✓ Enable] Use Data Item To Enable Disable

Interval: 100 msec

Name

The name identifies the Math & Logic device. It can be up to 50 characters long, may contain spaces, but must not begin with a space. It also may not contain any periods.

Description

This optional field further describes the Math & Logic device. It can be up to 255 characters long.

Simulate

Check this box to enable data simulation for all Math & Logic data items on this device. This provides a quick way to switch between real and simulated data for a large number of data items. Refer to the Cyberlogic OPC Server Help for a full discussion about simulating data.

Note If the Simulate checkbox is grayed-out, it indicates that simulation has already been selected at a higher level.

Disable Writes

Check this box to disable write requests for all data items at this level and below. By default, this box is not checked and writes are enabled.

Note If the Disable Writes checkbox is grayed-out, it indicates that writes have already been disabled a higher level.

Device Logic Enable

This radio selection allows you to enable or disable the Math & Logic programs for the data items in the device. When device logic is enabled, the Run Logic criteria will cause the programs to execute. When device logic is disabled, the programs will not run, regardless of the Run Logic criteria. You will configure the default settings for the Run Logic criteria on the Settings tab, but each data item can override the default.

If you select ***Enable***, device logic will be enabled.

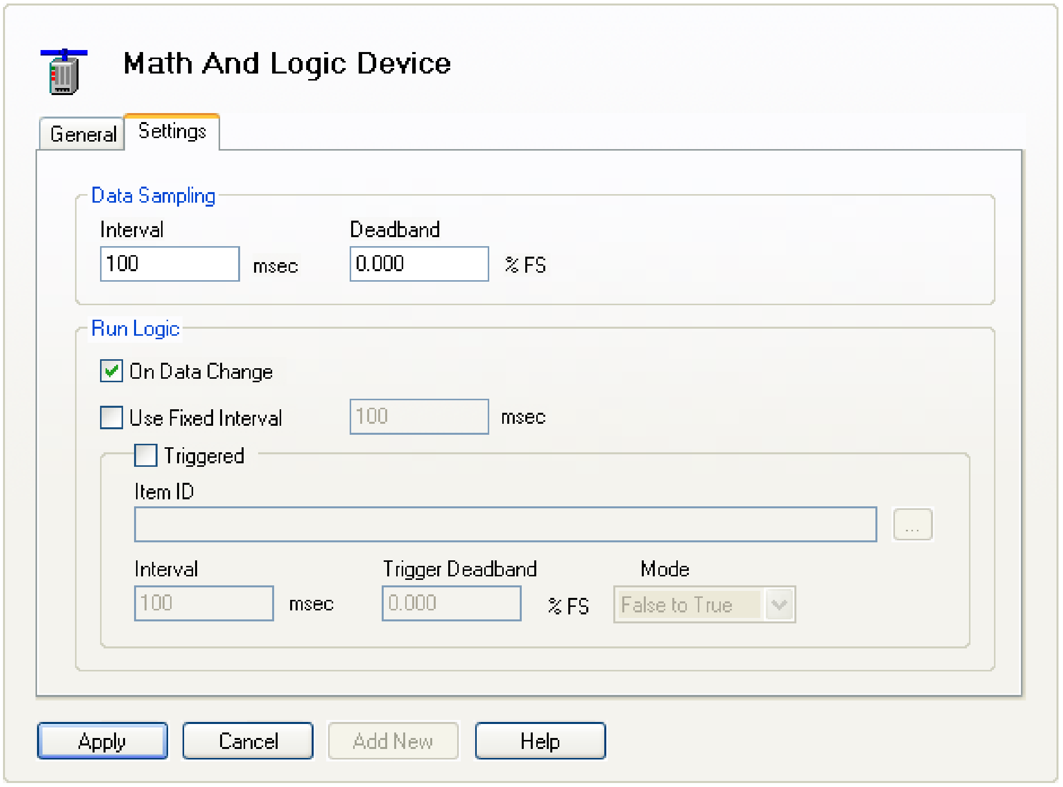
If you select ***Use Data Item To Enable***, you must specify a data item that will control the device logic. You must also enter a value for ***Interval*** to indicate how often the value of the data item will be checked. If the value of the specified data item is true, device logic will be enabled, otherwise it will be disabled. You can click the ***Browse...*** button to open a window that will allow you to browse for the desired data item.

Note If the item you choose contains numeric data, a value of zero is taken as false and a nonzero value is true.

If you select ***Disable***, device logic will be disabled.

Settings Tab

This tab is used to specify the default settings used by all data items under this device. Each data item can either use these settings, or define its own.


Math And Logic Device

General | Settings

Data Sampling
Interval: 100 msec
Deadband: 0.000 % FS

Run Logic
[✓] On Data Change
[ ] Use Fixed Interval 100 msec
[ ] Triggered
Item ID:
Interval: 100 msec
Trigger Deadband: 0.000 % FS
Mode: False to True

Data Sampling

Math & Logic programs use OPC data items as inputs. This section allows you to specify how often to check these data items for a change in their value. You do this by entering a sampling interval and a deadband. These are default values. Each data item declaration in a program can override these defaults, if desired. Refer to the Data Item Declarations section for information on how to do this.

Interval

You may specify the default interval, in milliseconds, at which you want the input data to be sampled. The ITEM variable declaration in your program can override this value.

Deadband

Here you may specify the default deadband in percent of full scale. This is the minimum amount that will be considered a change in the value. The deadband helps to eliminate problems caused by values that jitter. The ITEM variable declaration in your program can override this value, if desired. Refer to the Data Item Declarations section for information on how to do this.

Caution! In keeping with the OPC specifications, the deadband functions apply only to data items that have a dwEUType of Analog. No data items have this type by default. To convert a data item to Analog, you must apply a conversion to it. This allows you to associate engineering units with the data item, and it is the engineering units range that is used for the deadband calculation.

Run Logic

This section allows you to specify when to run the programs in the Math & Logic data items. There are three methods available, and you can choose any combination. If you choose more than one, the programs will run when any one of the conditions are met. This is a default setting. Each data item can override this default, if desired. Refer to the discussion of the Advanced... button on the Data Item Program tab for information on how to do this.

On Data Change

Check this box to run the programs whenever their input data values change.

If a program has more than one input, it will run when any input changes.

Note: Programs can exclude some inputs from triggering On Data Change execution. Refer to the Data Item Declarations section for information on how to do this.

Use Fixed Interval

Check this box to run the programs at the interval you specify.

Note If multiple Run Logic conditions are used, the Fixed Interval is restarted each time any of the Run Logic conditions triggers the program execution.

Triggered

Check this box to run the programs when the specified data item changes state.

If you choose this option, you must specify an Item ID, Mode and Interval. You may also specify a Trigger Deadband.

Item ID

Specify a data item to use as the trigger. When the value of the trigger data item changes state as specified in the Mode selection, the programs run. Click the ***Browse...*** button to open a window that will allow you to browse for the desired data item.

Interval

You must specify an interval when using triggered execution. This value specifies how often the trigger data item will be read to see if it has changed state.

Deadband

Here you may specify the deadband in percent of full scale. This is the minimum amount that will be considered a change in the value. The deadband helps to eliminate problems caused by values that jitter.

Caution!

In keeping with the OPC specifications, the deadband functions apply only to data items that have a dwEUType of Analog. No data items have this type by default. To convert a data item to Analog, you must apply a conversion to it. This allows you to associate engineering units with the data item, and it is the engineering units range that is used for the deadband calculation.

Mode

You must choose the type of change in the trigger item that will cause the logic to run.

- Choose ***False to True*** to trigger when the value changes from false to true.
- Choose ***True to False*** to trigger when the value changes from true to false.
- Choose ***Any Change*** to trigger when any change in the value occurs.

Note

In False to True or True to False mode:

If the item you choose contains numeric data, a value of zero is taken as false and a nonzero value is true.

In Any Change mode:

If the item you choose contains numeric or string data, any change in the value triggers the write.

Folders

A folder logically groups data items and other folders. You can place folders directly under devices or under other folders, up to four levels deep.

Caution!

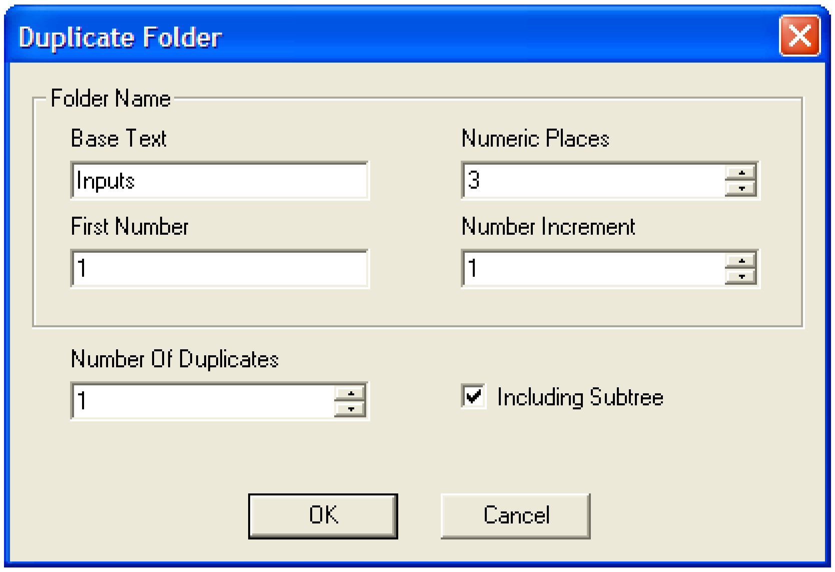
After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.

Creating a New Folder

Right-click on an existing device or folder, and select ***New*** and then ***Folder*** from the context menu.

Duplicating a Folder

You can create multiple folders in a single operation by duplicating an existing one. This can help you to quickly create similarly-configured folders. To duplicate a folder, right-click on it and select ***Duplicate...*** from the context menu.



Base Text: Inputs
Numeric Places: 3
First Number: 1
Number Increment: 1
Number Of Duplicates: 1
[✓] Including Subtree

The above dialog box opens. You must specify how the duplicates are to be named by entering values for the ***Base Text***, ***First Number***, ***Numeric Places*** and ***Number Increment*** fields. To name the duplicated folders, the editor begins with the base text and appends a number to it. The first duplicate uses the selected First Number value with the specified number of digits. The editor then adds Number Increment to that value for each of the remaining duplicates.

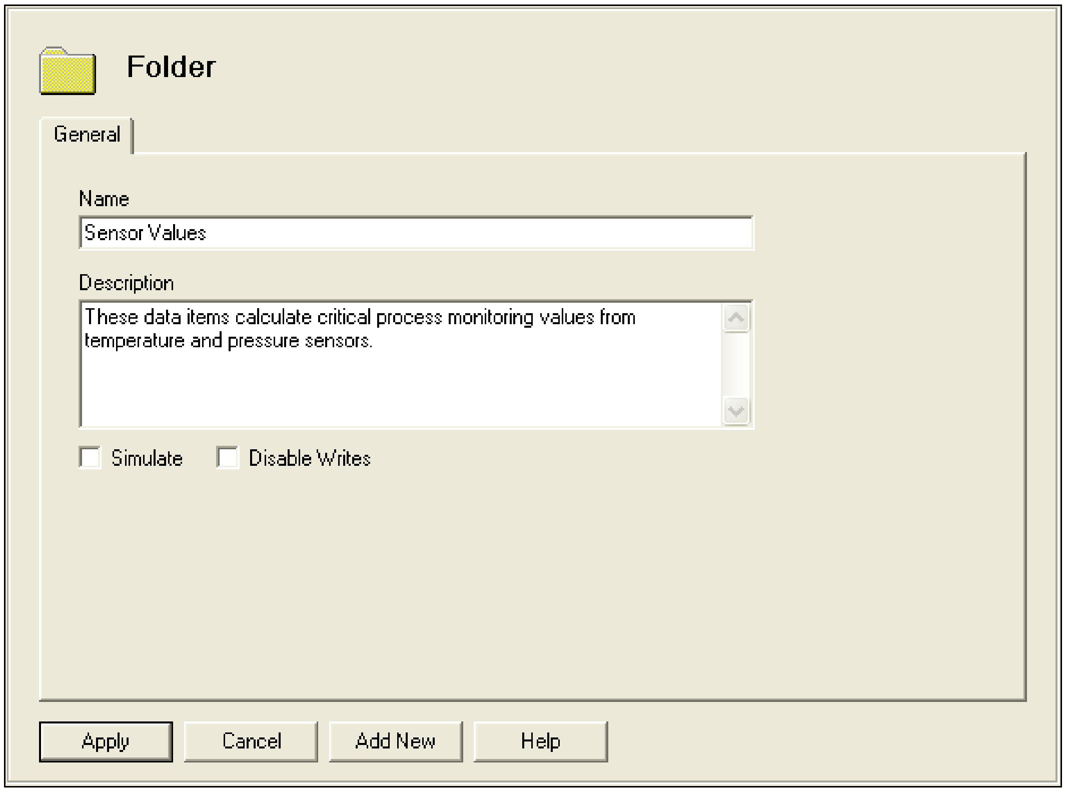
As an example, if Numeric Places is 3 and First Number is 2, the number 002 will be appended to the base text.

Use the ***Number Of Duplicates*** field to specify the number of folders you wish to create. If you want to duplicate all branches within the original folder, check the ***Including Subtree*** checkbox.

Deleting a Folder

To delete an existing folder, select it and press the ***Delete*** key, or right-click on the folder and select ***Delete*** from the context menu.

General Tab



Name: Sensor Values

Description: These data items calculate critical process monitoring values from temperature and pressure sensors.

☐ Simulate ☐ Disable Writes

Name

The Name identifies this folder. It can be up to 50 characters long, may contain spaces, but must not begin with a space. It also may not contain any periods.

Description

This optional field further describes the folder. It can be up to 255 characters long.

Simulate

Check this box to enable data simulation for all data items at this level or below. This provides a quick way to switch between real and simulated data for a large number of data items. Refer to the Cyberlogic OPC Server Help for a full discussion about simulating data.

Note If the Simulate checkbox is grayed-out, it indicates that simulation has already been selected at a higher level.

Disable Writes

Check this box to disable write requests for all data items at this level or below. By default, this box is not checked and writes are enabled.

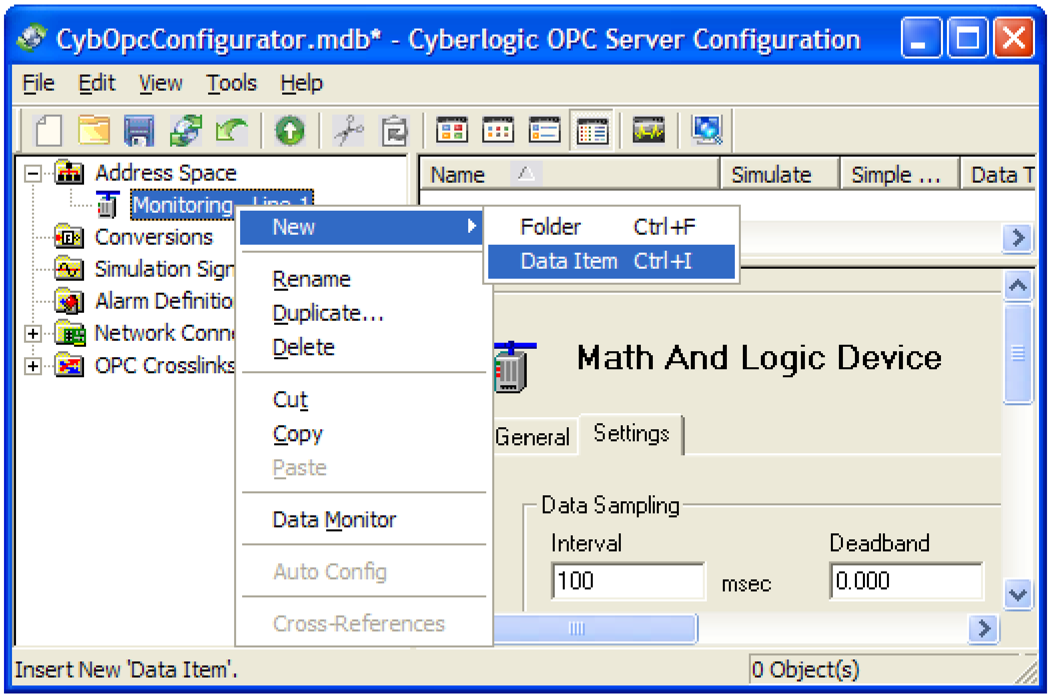
Note If the Disable Writes checkbox is grayed-out, it indicates that writes have already been disabled a higher level.

Math & Logic Data Items

Math & Logic data items contain the Math & Logic programs. They take the value that the program returns when it executes. You can place Math & Logic data items directly in a Math & Logic device, or in a folder within a Math & Logic device. Normally, the programs are enabled and run according to criteria you configure in the parent device. However, an advanced configuration feature allows you to modify the enable criteria and override the Run Logic criteria.

Caution! After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.

Creating a New Math & Logic Data Item



Interval: 100 msec
Deadband: 0.000

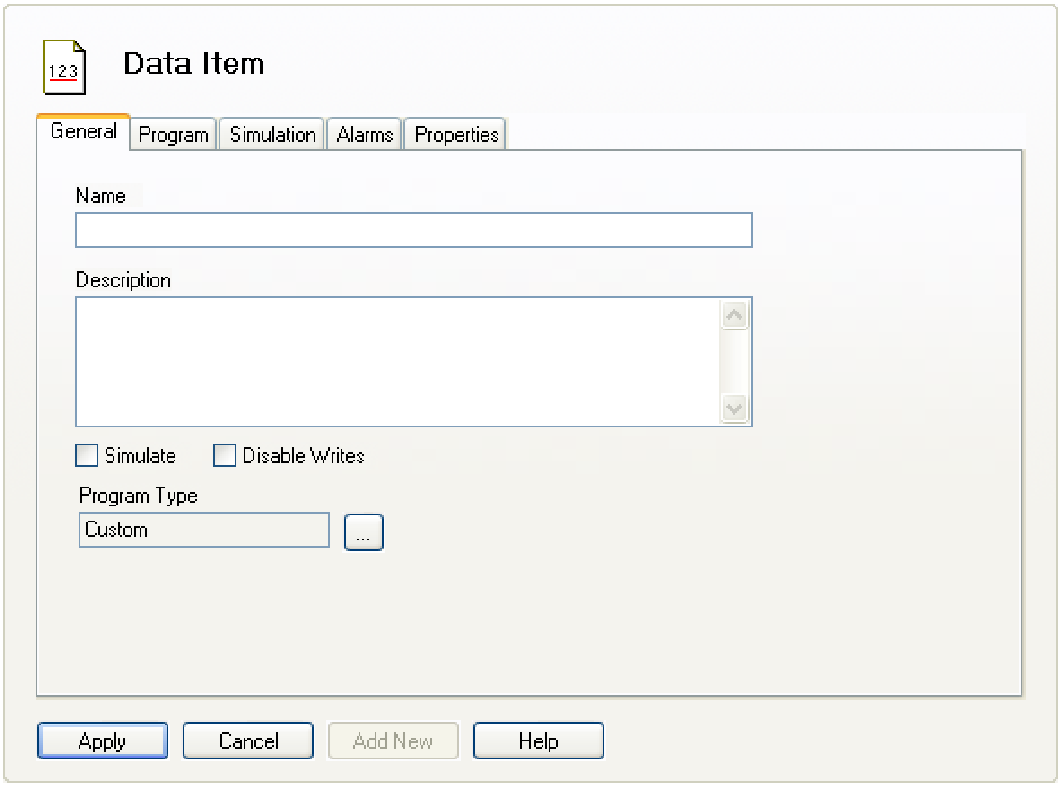
Insert New 'Data Item'.
0 Object(s)

Right-click on a Math & Logic device or a folder within a Math & Logic device. Select ***New*** and then ***Data Item*** from the context menu.

Deleting Math & Logic Data Items

To delete an existing Math & Logic data item, select it and press the ***Delete*** key, or right-click on the data item and select ***Delete*** from the context menu.

General Tab



Program Type: Custom

Name

The Name identifies the data item. It can be up to 50 characters long, may contain spaces, but must not begin with a space. It also may not contain any periods.

Description

This optional field further describes the data item. It can be up to 255 characters long.

Simulate

Check this box to enable data simulation for this data item. Refer to the Cyberlogic OPC Server Help for a full discussion about how to simulate data.

Note If the Simulate checkbox is grayed-out, it indicates that simulation has already been selected at a higher level.

Disable Writes

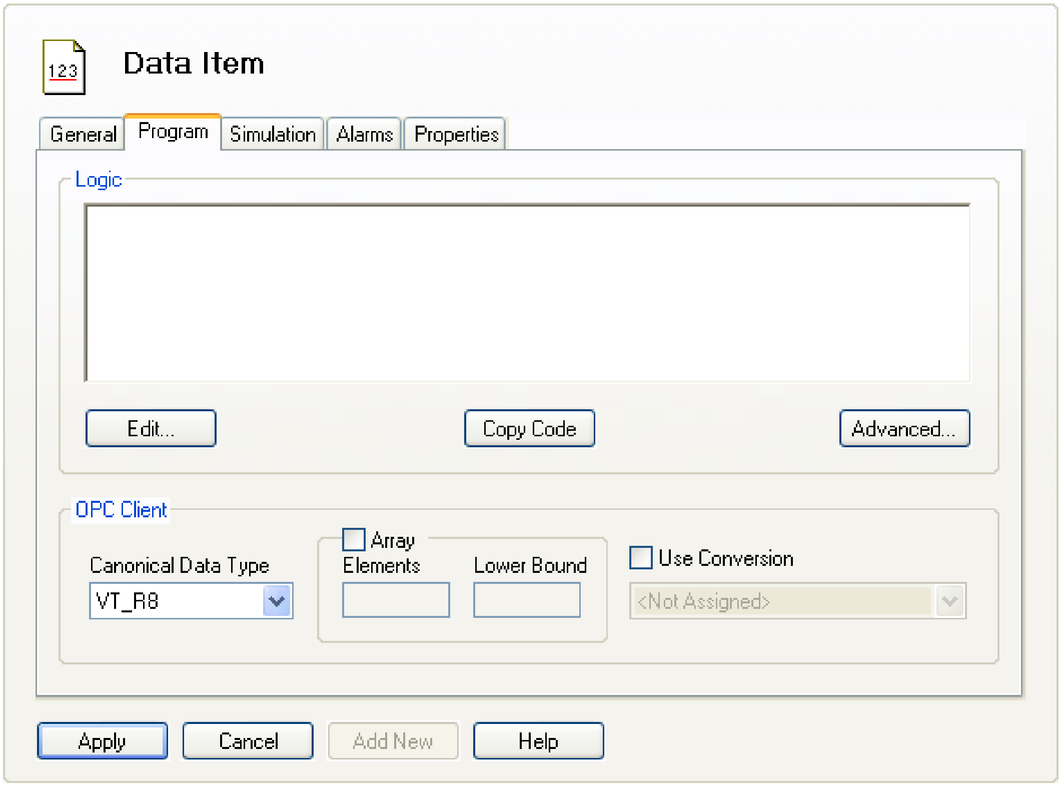
Check this box to disable all write requests for this data item. By default, this box is not checked and writes are enabled.

Note If the Disable Writes checkbox is grayed-out, it indicates that writes have already been disabled a higher level.

Program Type

Here you may select the type of program you want to use for the data item. The default is *Custom*. For detailed information on this field and the available program types, refer to the Math & Logic Program Types section.

Program Tab



Canonical Data Type: VT_R8

Logic

This section is where you view and edit the data item's Math & Logic program. The large field provides a display-only view of the program.

Note If the background color for the Logic window is red, it indicates that the program has not yet been compiled, and therefore no executable code exists.

Edit...

Click this button to open the Math & Logic Editor. This is where you edit, compile and debug the program.

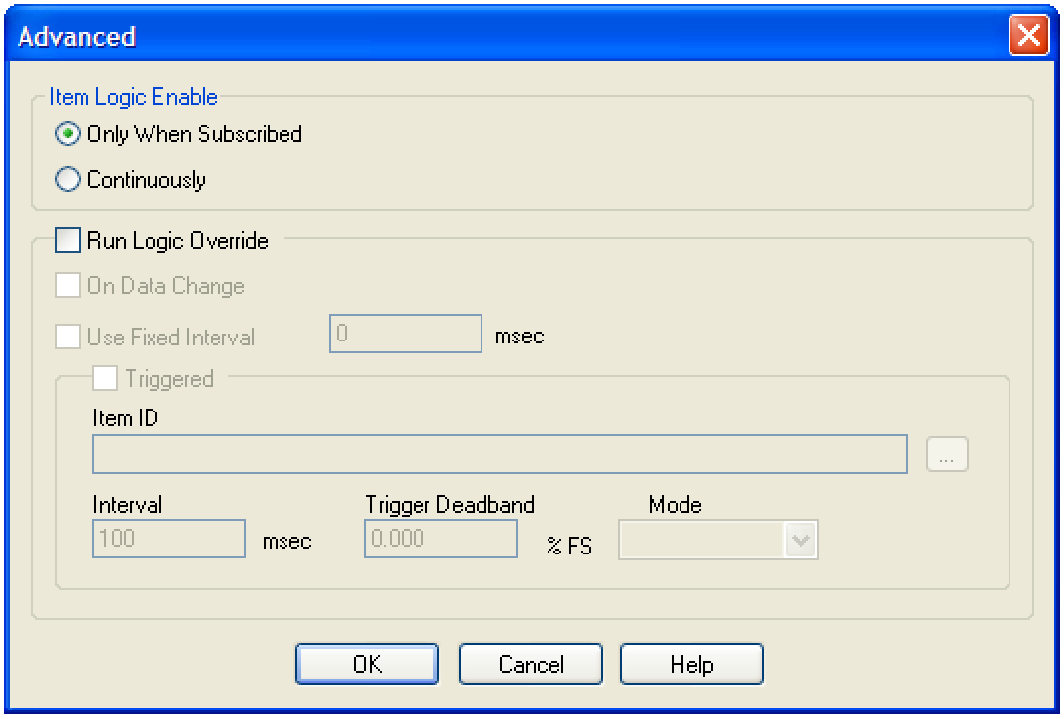
Note You can also start the Math & Logic Editor by double-clicking on the selected data item in the address space tree.

Copy Code

Click this button to copy the program code. You can then paste the code into another Math & Logic data item and then modify it in the new data item.

Advanced...

Click this button to open the Advanced editing window. Normally, the criteria that enable and run the program are configured in the Math & Logic device. This window allows advanced users to modify these criteria. You can modify the conditions that enable the logic, and override the criteria that cause it to run.



By default, the program runs only when at least one client subscribes to the data item. This reduces the load on the processor. However, the Item Logic Enable selection lets you request that the logic run continuously, even without any subscriptions.

The Run Logic Override section lets you declare that the run criteria in the Math & Logic device should be ignored, and the criteria you specify here should be used instead. For details on editing these criteria, refer to the Math & Logic device Settings Tab discussion.

OPC Client

This section allows you to configure how the program's result will be presented to the OPC client software.

Canonical Data Type

Select the data type you want to use to present the results to the client. The default is VT_R8 (floating-point double).

Array

Check this box to specify that the returned data will be in the form of an array.

Elements

Enter the number of elements in the array.

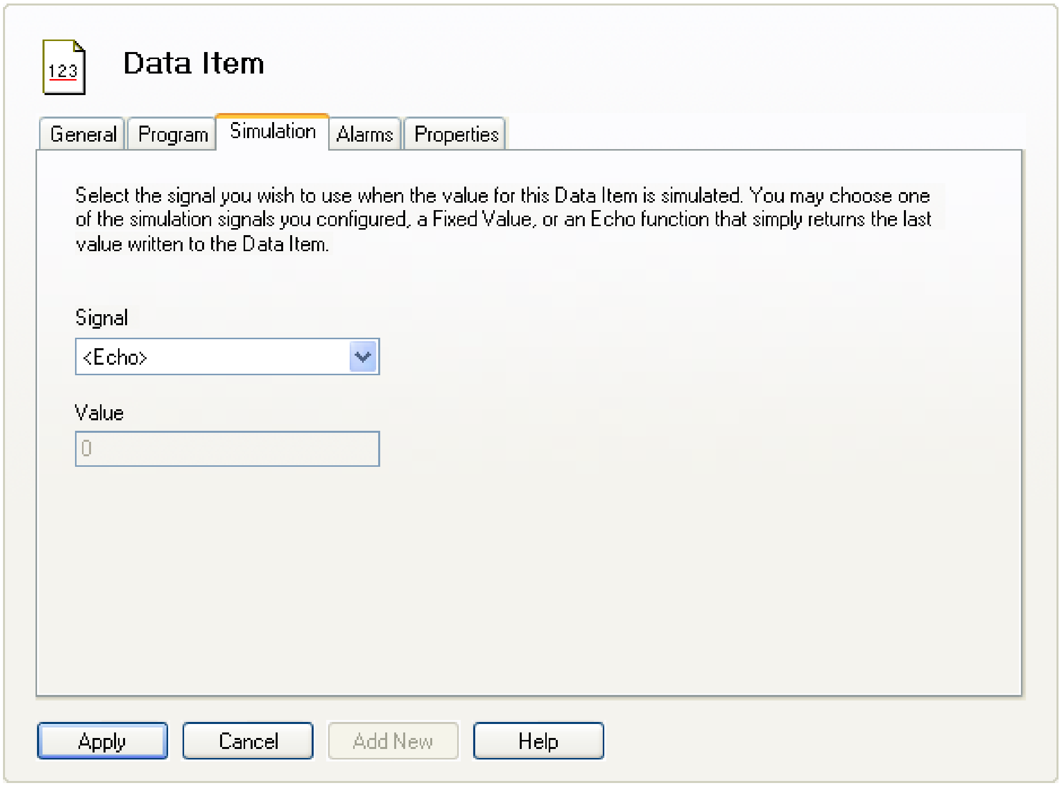
Lower Bound

Enter the lowest element number for the array.

Use Conversion

Check this box to apply a conversion to the data before it is presented to the client. You must then select the conversion you want to use from the list of defined Conversions.

Simulation Tab



Select the signal you wish to use when the value for this Data Item is simulated. You may choose one of the simulation signals you configured, a Fixed Value, or an Echo function that simply returns the last value written to the Data Item.

Signal

<Echo>

Value

0

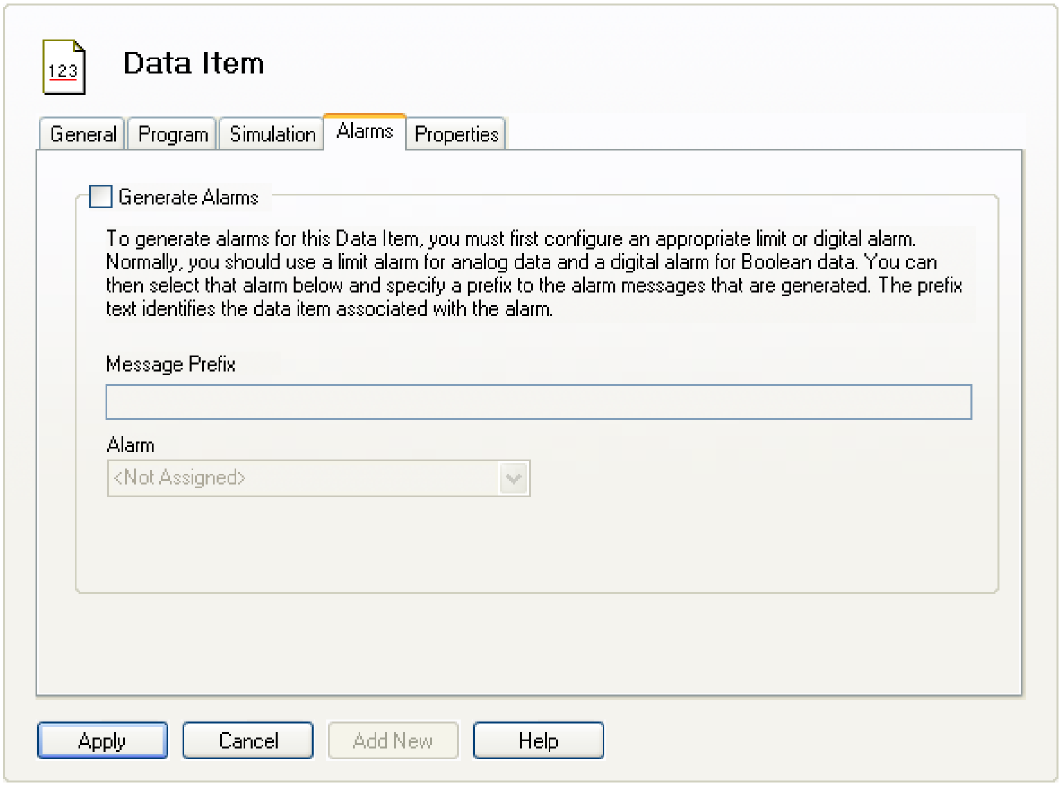
Signal

If you enabled simulation on the General tab or at a higher level, you must select how to simulate the data item's value. Your choices are: a fixed value, an echo of the last value written to the item, or one of the previously-defined Simulation Signals.

Value

When simulation is enabled and the Signal field is set to Fixed Value, the data item will be set to this value.

Alarms Tab



Generate Alarms

To generate alarms for this Data Item, you must first configure an appropriate limit or digital alarm. Normally, you should use a limit alarm for analog data and a digital alarm for Boolean data. You can then select that alarm below and specify a prefix to the alarm messages that are generated. The prefix text identifies the data item associated with the alarm.

Message Prefix

Alarm

<Not Assigned>

Generate Alarms

If this box is checked, the server will test the alarm conditions for this data item. It will then generate alarms as appropriate.

Note Only OPC clients that support the OPC Alarms & Events specification can receive alarms.

Refer to the Cyberlogic OPC Server Help for a full discussion about how to create and use alarms.

Message Prefix

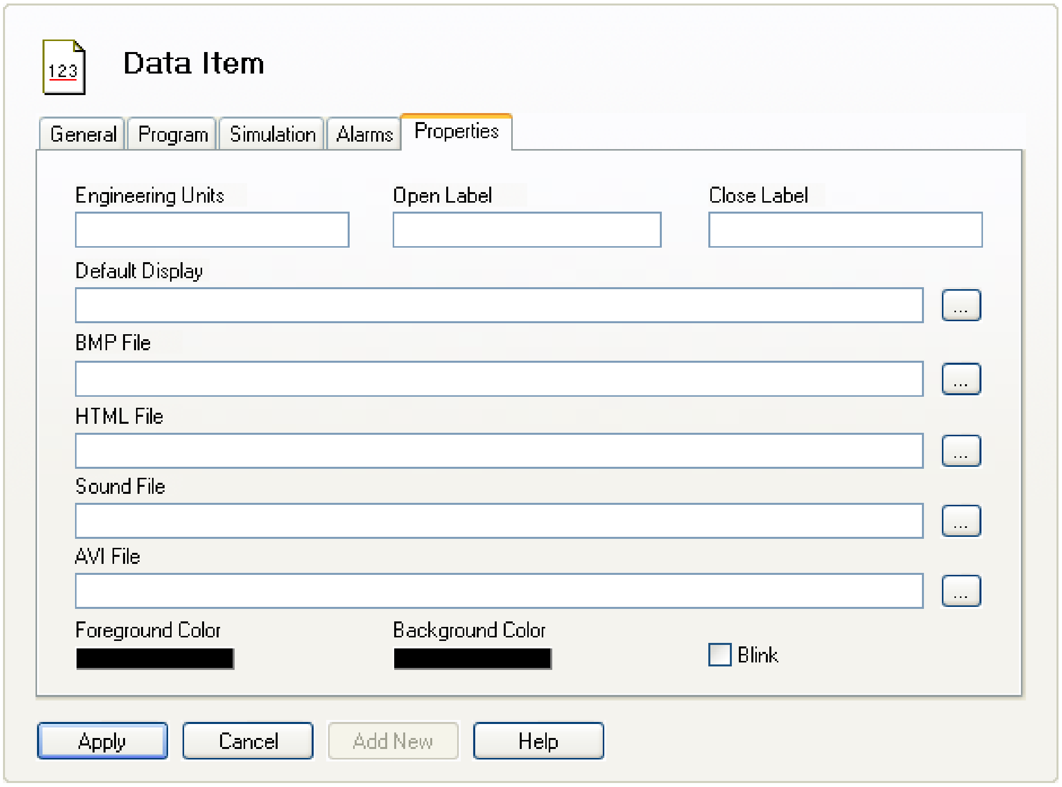
Enter the text for the first part of the alarm message. The second part will be the body text of the specific alarm that is generated.

Alarm

Select one of the previously-defined Alarm Definitions to serve as the alarm template for this data item.

Properties Tab

In addition to the main data item properties—value, quality and timestamp—the OPC specification includes several optional properties that your client application may use. This tab allows you to set these data item properties. These properties are static and do not change while the server is running.



Engineering Units

This is OPC property ID 100. It specifies the engineering units text, such as DEGC or GALLONS. It can be up to 50 characters long.

Open Label

This is OPC property ID 107, and is presented only for discrete data. This text describes the contact when it is in the open (zero) state, such as STOP, OPEN, DISABLE or UNSAFE. It can be up to 50 characters long.

Close Label

This is OPC property ID 106, and is presented only for discrete data. This text describes the contact when it is in the closed (non-zero) state, such as RUN, CLOSE, ENABLE or SAFE. It can be up to 50 characters long.

Default Display

This is OPC property ID 200. It is the name of an operator display associated with this data item. It can be up to 255 characters long.

BMP File

This is OPC property ID 204. It is the name of a bitmap file associated with this data item, for example C:\MEDIA\FIC101.BMP. It can be up to 255 characters long.

HTML File

This is OPC property ID 206. It is the name of the HTML file associated with this data item, for example http://mypage.com/FIC101.HTML. It can be up to 255 characters long.

Sound File

This is OPC property ID 205. It is the name of the sound file associated with this data item, for example C:\MEDIA\FIC101.WAV. It can be up to 255 characters long.

AVI File

This is OPC property ID 207. It is the name of the AVI file associated with this data item, for example C:\MEDIA\FIC101.AVI. It can be up to 255 characters long.

Foreground Color

This is OPC property ID 201. Click on the box and select the foreground color used to display the item.

Background Color

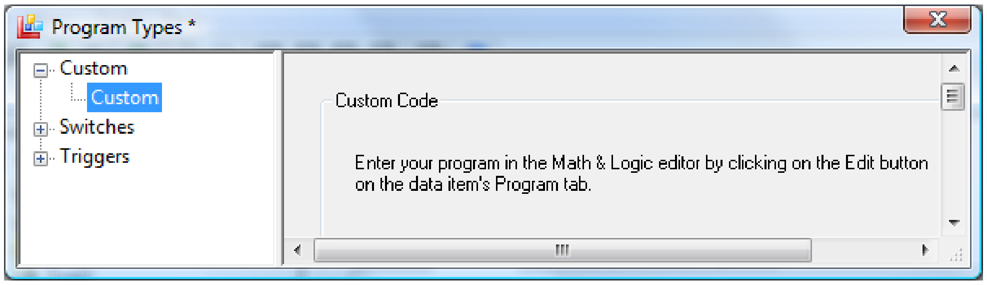
This is OPC property ID 202. Click on the box and select the background color used to display the item.

Blink

This is OPC property ID 203. Check this box to indicate that displays of the item should blink.

Math & Logic Program Types

When you create a Math & Logic data item, you must select a program type on its General Tab.



Enter your program in the Math & Logic editor by clicking on the Edit button on the data item's Program tab.

There are three groups of choices:

- Custom program type lets you write your own programs. This type is fully functional only in Premier Suite and Enterprise Suite products. In other Cyberlogic OPC products, you can create and edit custom data items, but they run in demo mode only. For details, refer to Pre-Programmed and Demo Math & Logic Items in the Introduction section.
- Switches include pre-programmed applets that return true or false according to a specific condition. These types are fully functional in all Cyberlogic OPC products.
- Triggers include pre-programmed applets that increment each time the specified time events occur. Some triggers respond to time intervals, while others use the system time and date. These types are fully functional in all Cyberlogic OPC products.

Note You can copy a pre-programmed applet’s generated program by pressing the Copy Code button on the Program tab. You can then paste it into your own Custom program and modify it as needed.

Custom Programs

Math & Logic programs of type Custom are fully functional only in Cyberlogic’s Premier Suite and Enterprise Suite products. In other Cyberlogic OPC products, you can create and edit custom data items, but they run in demo mode only. For details, refer to Pre-Programmed and Demo Math & Logic Items in the Introduction section.

When you select a custom type, the program is initially empty, and you use the Math & Logic Editor to create the desired program.

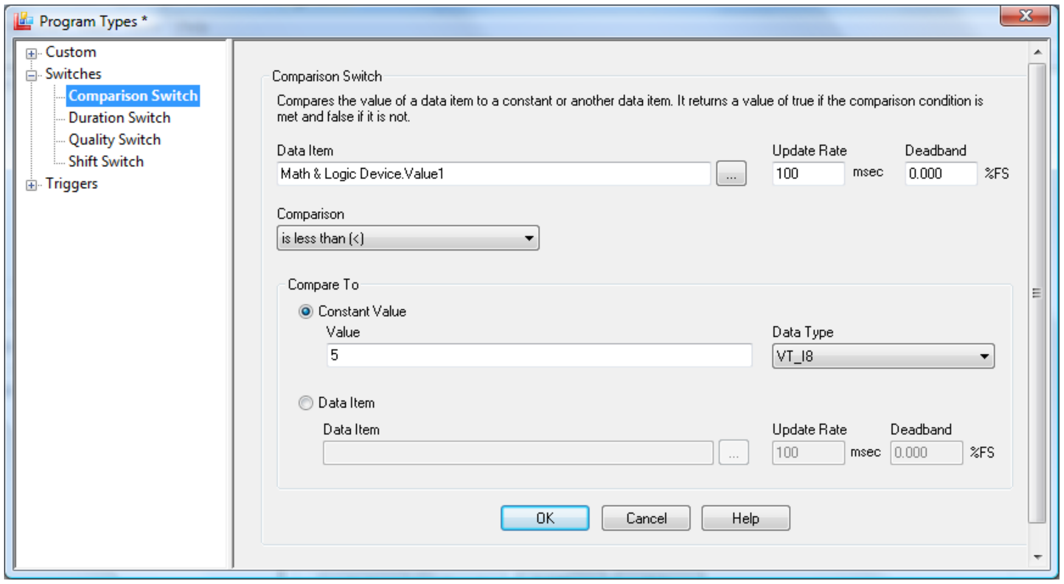
Switches

Math & Logic applets located in the Switches group are fully functional in all Cyberlogic OPC products. You may choose one of several pre-programmed applets, each of which evaluates a specific condition and then returns a true or false value. Each switch also has a public variable named "Not" whose value is the opposite of the switch's value. You cannot modify the switch's program, but you can specify certain parameters that it uses.

The pre-programmed choices are:

Comparison Switch

This applet compares the value of a data item to a constant or another data item. It returns a value of true if the comparison condition is met and false if it is not.



Comparison Switch

Compares the value of a data item to a constant or another data item. It returns a value of true if the comparison condition is met and false if it is not.

Data Item: Math & Logic Device.Value1
Update Rate: 100 msec
Deadband: 0.000 %FS

Comparison: is less than (<)

Compare To
Constant Value
Value: 5
Data Type: VT_I8

Data Item
Data Item:
Update Rate: 100 msec
Deadband: 0.000 %FS

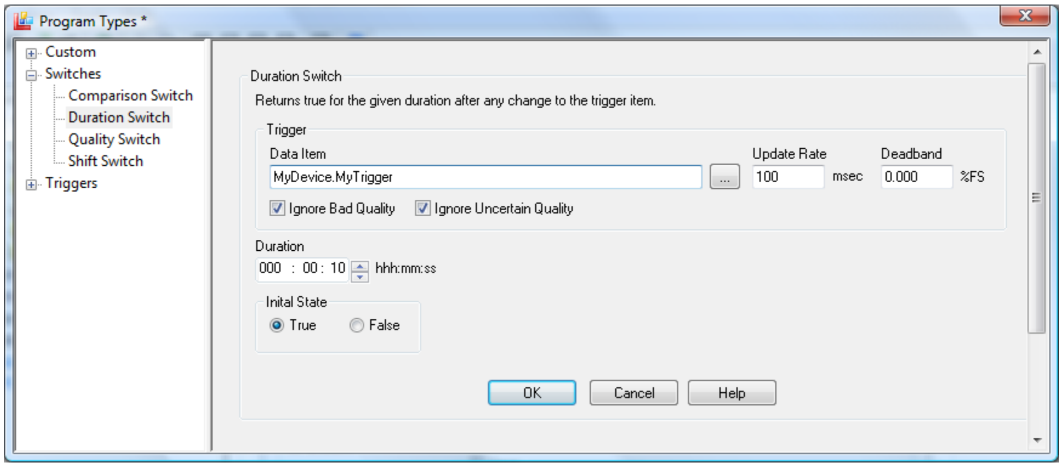
The parameters are:

Data Item:	ItemID of the data item you want to compare
Update Rate:	The interval at which you want to update the value of the data item, in milliseconds
Deadband:	The deadband in percent of full scale
Comparison:	The type of comparison: is greater than (>), is greater than or equal to (>=), is equals (==), is not equal to (!=), is less than or equal to (<=), is less than (<), logical AND, logical OR
Constant Value:	Select this to compare the data item to a constant value, which you must then specify using the *Value* and *Data Type* fields
Data Item:	Select this to compare the data item to another data item, which you must then specify using the *Data Item ID, Update Rate* and *Deadband* fields

Note The Comparison type includes the logical AND and logical OR. When the public variable "Not" is used, that also provides the NAND and NOR functionality. By chaining multiple AND/OR/NAND/NOR type data items, any boolean expression can be created. However, the same boolean expression, when implemented by a single custom program, will run much faster.

Duration Switch

This applet returns true for the given duration following any change to the Trigger item.



Duration Switch

Returns true for the given duration after any change to the trigger item.

Trigger

Data Item: MyDevice.MyTrigger

Update Rate: 100 msec

Deadband: 0.000 %FS

[✓] Ignore Bad Quality [✓] Ignore Uncertain Quality

Duration: 000 : 00 : 10 hhh:mm:ss

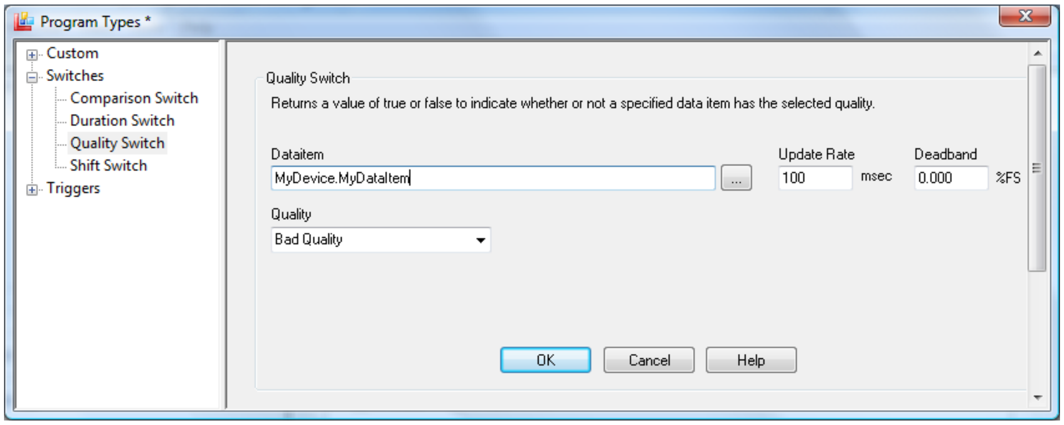
Inital State: [✓ True] False

The parameters are:

Data Item:	ItemID of the trigger data item
Update Rate:	The interval at which you want to update the value of the trigger data item, in milliseconds
Deadband:	The deadband in percent of full scale
Ignore Bad Quality:	Ignore trigger values with BAD quality if checked
Ignore Uncertain Quality:	Ignore trigger values with UNCERTAIN quality if checked
Duration:	Duration of the true output following any change to the trigger item
Initial State:	Initial state for the output (true or false)

Quality Switch

This applet returns a value of true or false to indicate whether or not a specified data item has the selected quality.



Quality Switch

Returns a value of true or false to indicate whether or not a specified data item has the selected quality.

Dataitem: MyDevice.MyDataItem

Update Rate: 100 msec

Deadband: 0.000 %FS

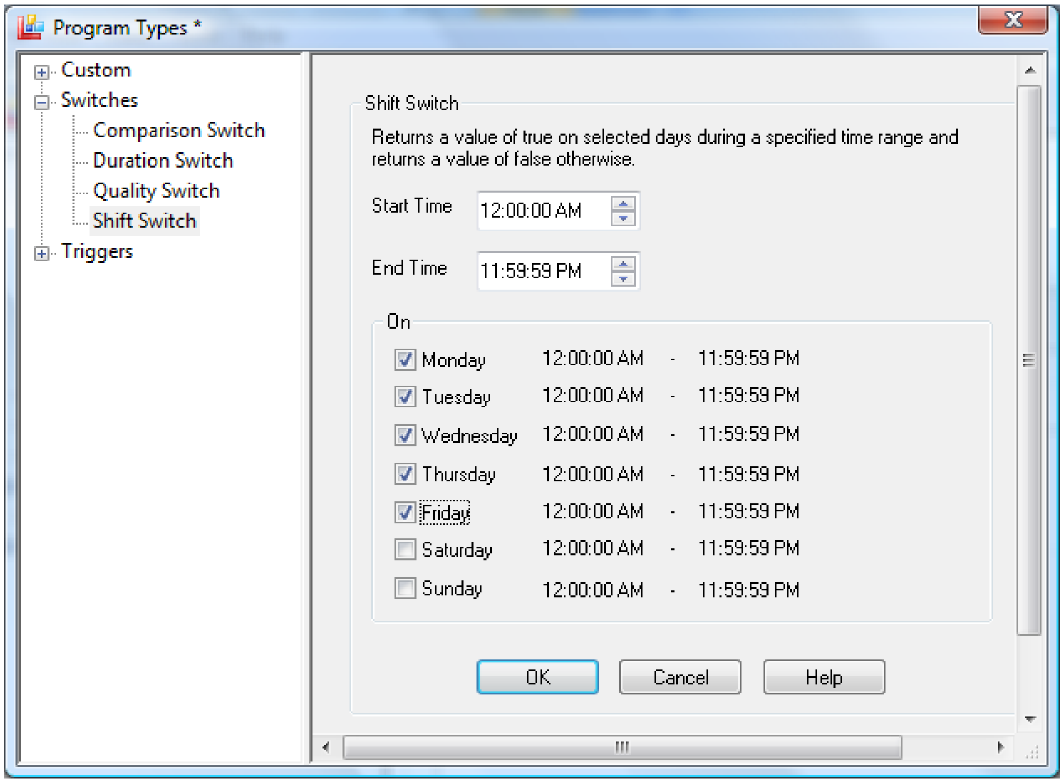
Quality: Bad Quality

The parameters are:

Data Item:	ItemID of the data item you want to test
Update Rate:	The interval at which you want to check the quality of the data item. The value is in milliseconds.
Deadband:	The deadband in percent of full scale
Quality:	Selects what quality of the data item should return a value of true. The valid choices are: Bad Quality, Not Bad Quality, Good Quality, Not Good Quality, Uncertain Quality, Not Uncertain Quality

Shift Switch

This applet returns a value of true on selected days during a specified time range, and returns a value of false otherwise.



Shift Switch

Returns a value of true on selected days during a specified time range and returns a value of false otherwise.

Start Time 12:00:00 AM

End Time 11:59:59 PM

On

✓ Monday	12:00:00 AM	-	11:59:59 PM
✓ Tuesday	12:00:00 AM	-	11:59:59 PM
✓ Wednesday	12:00:00 AM	-	11:59:59 PM
✓ Thursday	12:00:00 AM	-	11:59:59 PM
✓ Friday	12:00:00 AM	-	11:59:59 PM
☐ Saturday	12:00:00 AM	-	11:59:59 PM
☐ Sunday	12:00:00 AM	-	11:59:59 PM

The parameters are:

Start time:	The time of day that is the beginning of the period during which the value will be true
End time:	The time of day that is the end of the period during which the value will be true
On:	The days of the week on which the value will be true during the specified time period

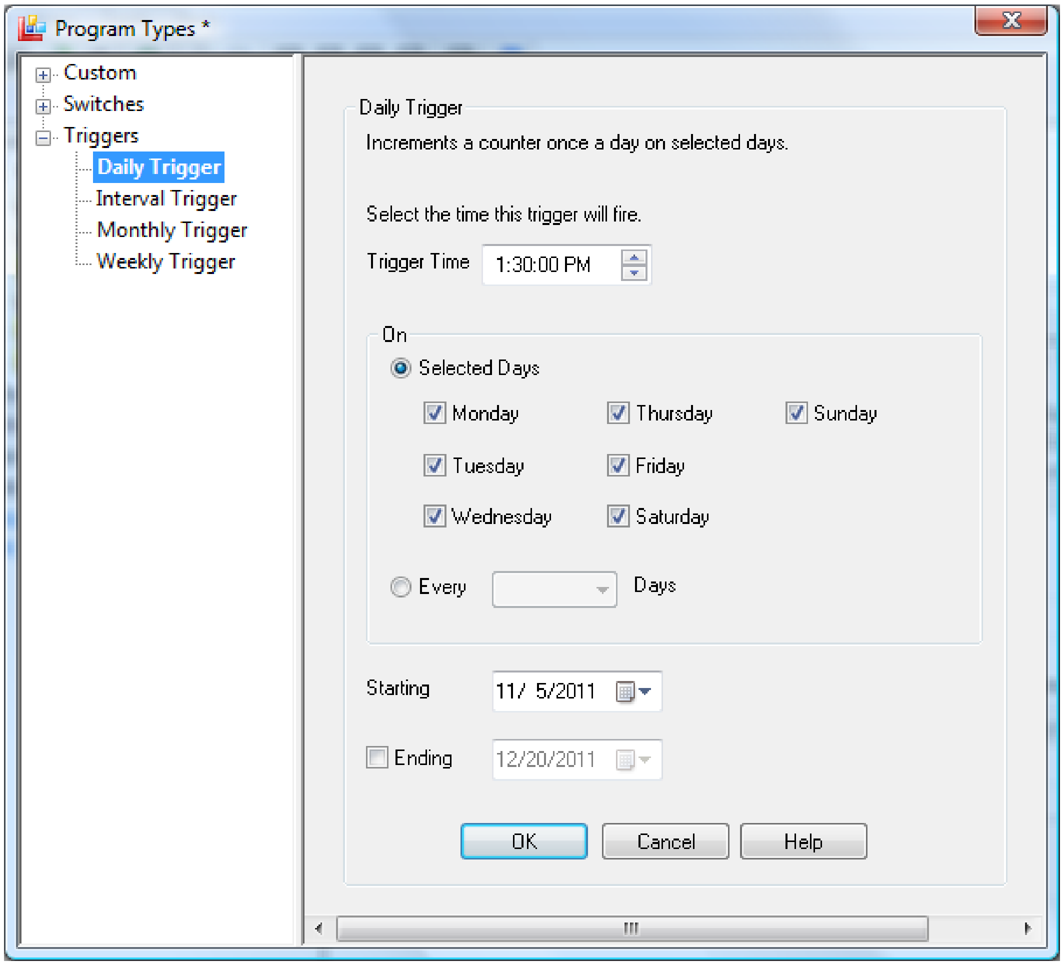
Triggers

Math & Logic applets located in the Triggers group are fully functional in all Cyberlogic OPC products. Their value increments each time the specified time events occur. Some triggers respond to time intervals, while others use the system time and date. You cannot modify the trigger's program, but you can specify certain parameters that it uses.

The pre-programmed choices are:

Daily Trigger

This applet increments a counter once a day on selected days.



Daily Trigger

Increments a counter once a day on selected days.

Select the time this trigger will fire.

Trigger Time 1:30:00 PM

On

(●) Selected Days

[✓] Monday [✓] Thursday [✓] Sunday

[✓] Tuesday [✓] Friday

[✓] Wednesday [✓] Saturday

( ) Every Days

Starting 11/ 5/2011

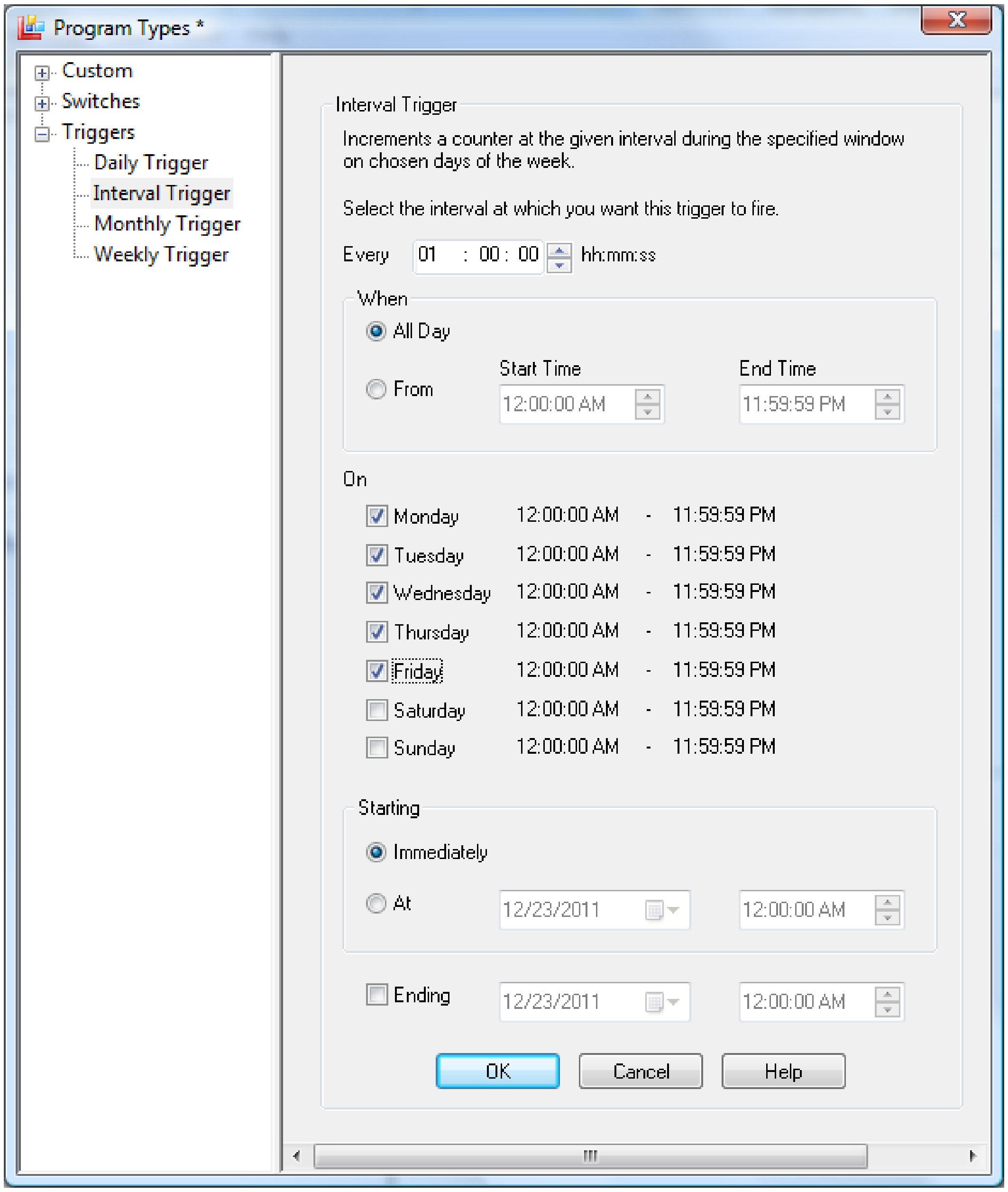
[ ] Ending 12/20/2011

The parameters are:

Trigger time:	The time of day at which the counter will increment
Selected Days:	Select this to indicate the days of the week on which the counter will increment. You must then select one or more days of the week.
Every:	Select this to specify an interval of days at which the counter will increment. You must then specify a number of days.
Starting:	The first date on which the counter will increment
Ending:	If checked, the last date on which the counter will increment

Interval Trigger

This applet increments a counter at specified interval during the specified window on chosen days of the week. The maximum interval is 24 hours.



Interval Trigger

Increments a counter at the given interval during the specified window on chosen days of the week.

Select the interval at which you want this trigger to fire.

Every 01 : 00 : 00 hh:mm:ss

When

(●) All Day

( ) From Start Time 12:00:00 AM End Time 11:59:59 PM

On

✓ Monday	12:00:00 AM	-	11:59:59 PM
✓ Tuesday	12:00:00 AM	-	11:59:59 PM
✓ Wednesday	12:00:00 AM	-	11:59:59 PM
✓ Thursday	12:00:00 AM	-	11:59:59 PM
✓ Friday	12:00:00 AM	-	11:59:59 PM
☐ Saturday	12:00:00 AM	-	11:59:59 PM
☐ Sunday	12:00:00 AM	-	11:59:59 PM

Starting

(●) Immediately

( ) At 12/23/2011 12:00:00 AM

☐ Ending 12/23/2011 12:00:00 AM

The parameters are:

Every:	The interval of time at which the counter will increment
All Day:	Select this if the counter should increment for the entire day
From:	Select this to specify a window of time during the day when the counter should increment
On:	The days of the week on which the counter will increment

Immediately:	Select this to begin incrementing the counter at the specified interval from now
At:	Select this to begin incrementing the counter at the specified interval from the selected time
Ending:	If checked, the last time the counter will increment

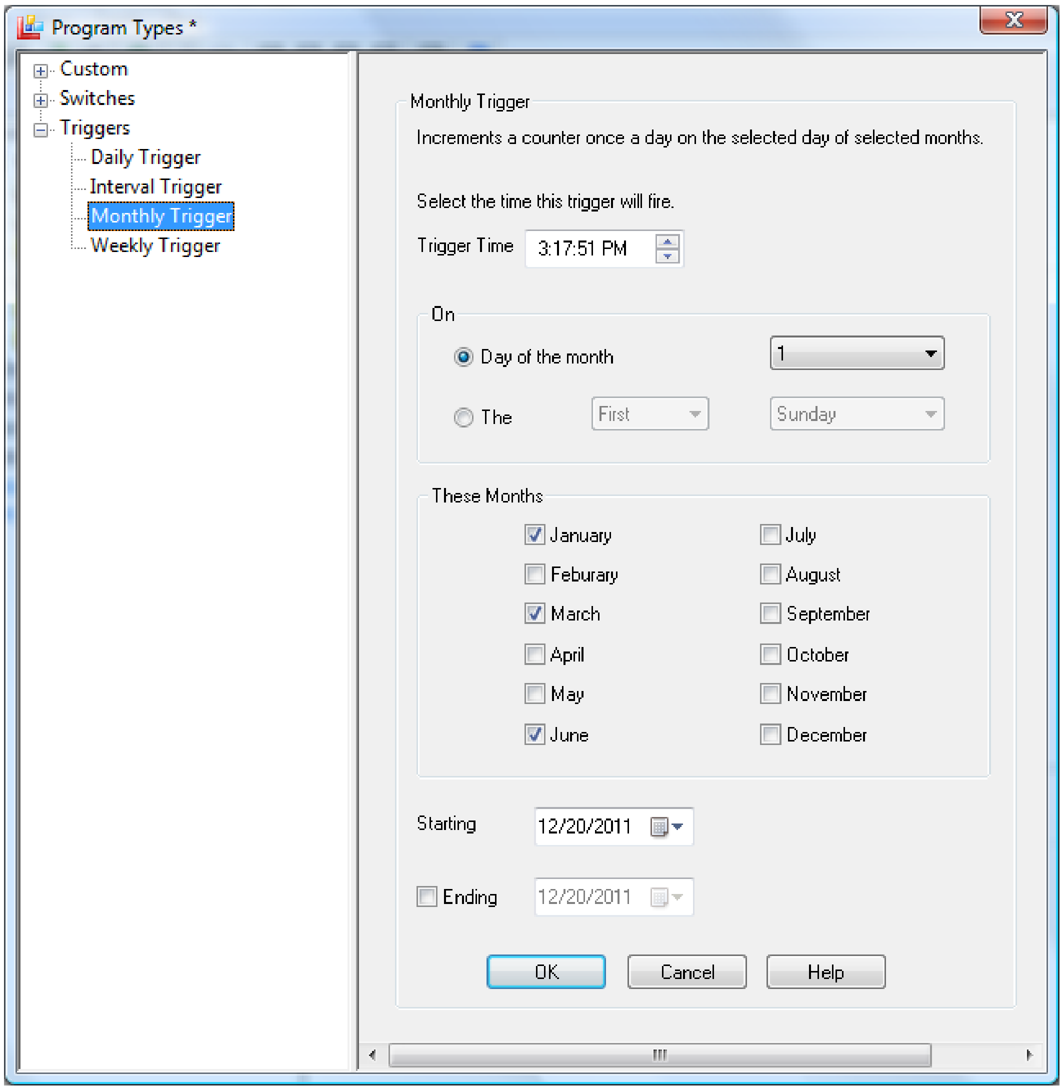
Examples:

The interval is 15 minutes and the current time is 9:03:16. If you choose *Immediately*, the increments will occur at 9:18:16, then at 9:33:16, then at 9:48:16, and so on. If you choose *At* 9:00:00, the increments will occur at 9:15:00, then at 9:30:00, then at 9:45:00, and so on.

The interval is 10 seconds and the current time is 9:03:16. If you choose *Immediately*, the increments will occur at 9:03:26, then at 9:03:36, then at 9:03:46, and so on. If you choose *At* 9:00:00, the increments will occur at 9:03:20, then at 9:03:30, then at 9:03:40, and so on.

Monthly Trigger

This applet increments a counter once a day on the selected day of selected months.



Monthly Trigger

Increments a counter once a day on the selected day of selected months.

Select the time this trigger will fire.

Trigger Time: 3:17:51 PM

On

[✓ Day of the month] 1

The First Sunday

These Months

✓ January	July
Feburary	August
✓ March	September
April	October
May	November
✓ June	December

Starting: 12/20/2011

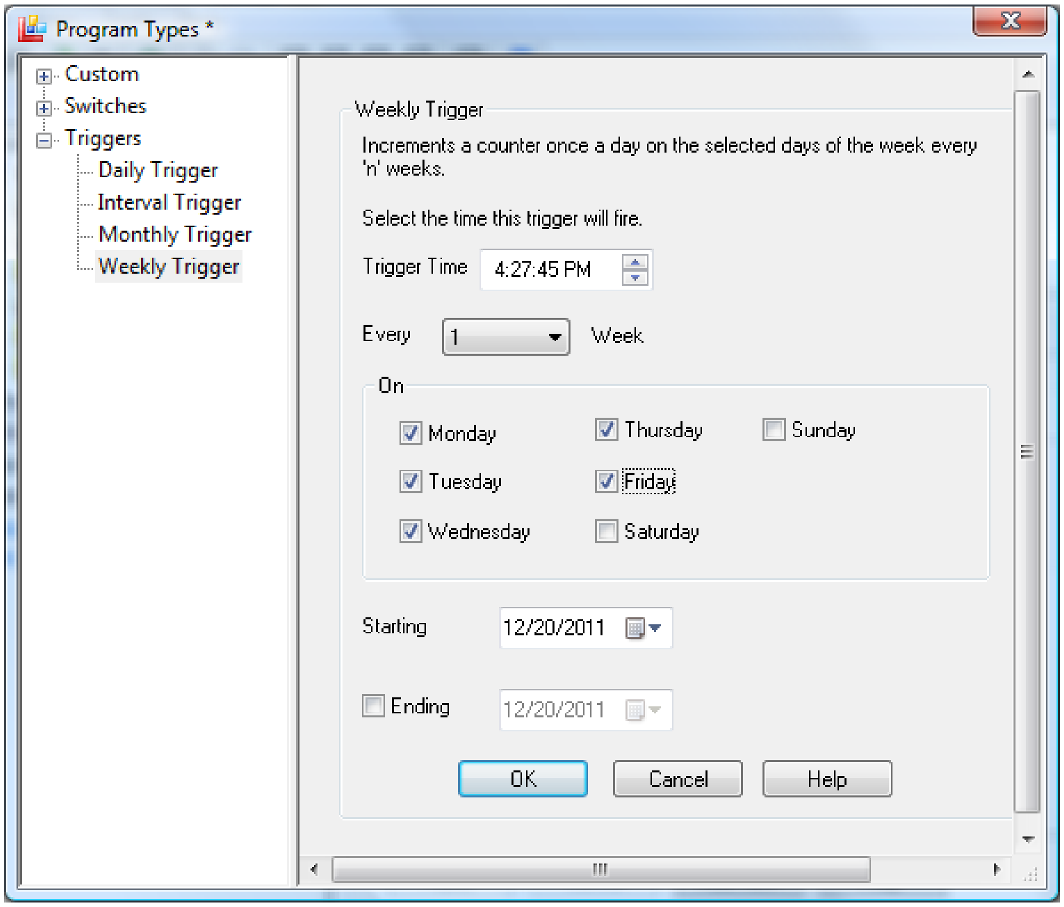
Ending: 12/20/2011

The parameters are:

Trigger time:	The time of day at which the counter will increment
Day of the month:	Select this to indicate the numeric day of the month on which the counter will increment
The:	Select this to indicate the instance of a weekday on which the counter will increment
These Months:	Select one or more months in which the counter will increment
Starting:	The first date on which the counter will increment
Ending:	If checked, the last date on which the counter will increment

Weekly Trigger

This applet increments a counter once a day on selected days of the week every so many weeks.


Program Types *

Custom
Switches
Triggers
Daily Trigger
Interval Trigger
Monthly Trigger
Weekly Trigger

Weekly Trigger

Increments a counter once a day on the selected days of the week every 'n' weeks.

Select the time this trigger will fire.

Trigger Time 4:27:45 PM

Every 1 Week

On

[✓] Monday [✓] Thursday [ ] Sunday
[✓] Tuesday [✓] Friday
[✓] Wednesday [ ] Saturday

Starting 12/20/2011

[ ] Ending 12/20/2011

The parameters are:

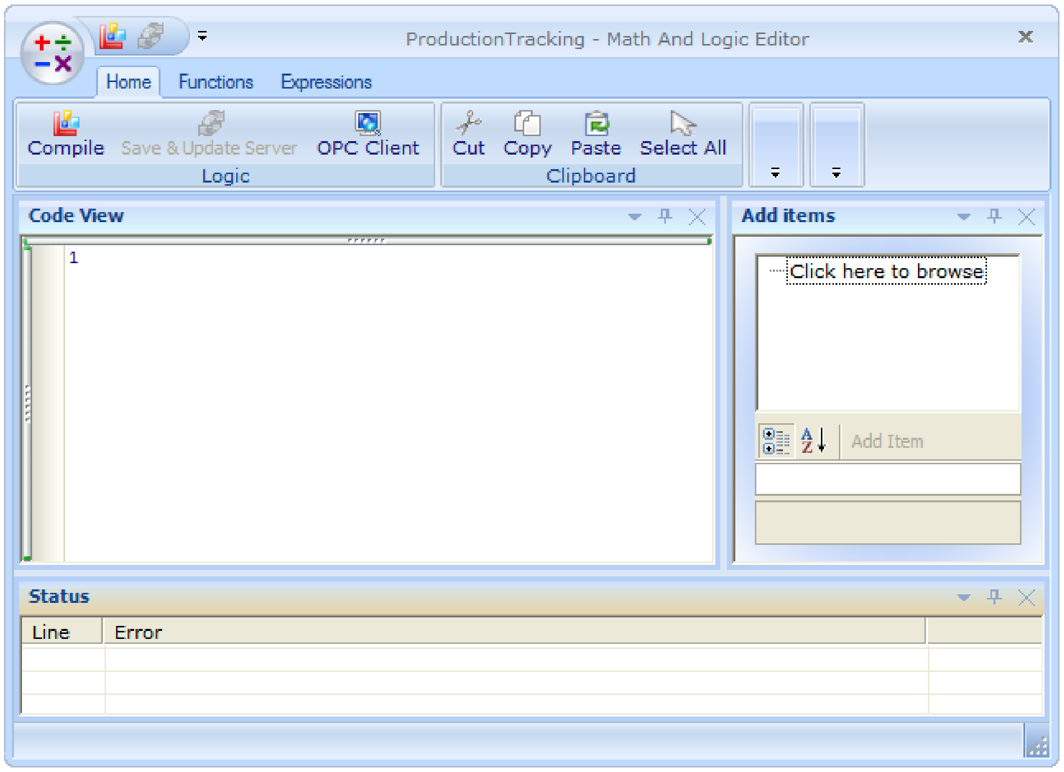
Trigger time:	The time of day at which the counter will increment
Every:	The interval of weeks at which the counter will increment
On:	The day(s) on which the counter will increment
Starting:	The first date on which the counter will increment
Ending:	If checked, the last date on which the counter will increment

Math & Logic Editor

The Math & Logic Editor allows you to create, edit and compile your programs using Cyberlogic's C-logic programming language. (Refer to Appendix A: C-logic Language

Reference for detailed information on programming in C-logic.) To launch the editor, select a Math & Logic data item and go to its ***Program*** tab, and then click the ***Edit...*** button. You can also launch the editor by double-clicking on the data item in the address space tree.

Note To get help with any function keyword or symbol in the C-logic editor, just put the cursor on the function name or keyword, or select the symbol, and press <F1>.



The editor has three panes for viewing and editing:

- Code View Pane
- Add Items Pane
- Status Pane

In addition, it has three ribbons containing the editing tools:

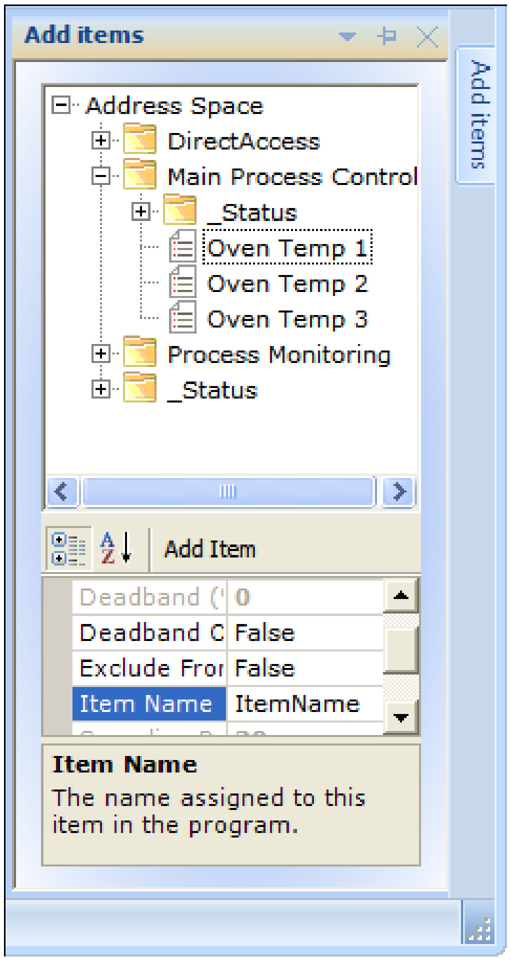
- Home Ribbon
- Functions Ribbon
- Expressions Ribbon

Code View Pane

The Code View pane is the main working area of the editor. This is where you edit your program, either by adding items from the ribbon, or by direct text editing.

Add Items Pane

The Add Items pane will assist you in adding data items as inputs or outputs for your program. To open it, click on the ***Add Items*** button at the right edge of the editor.



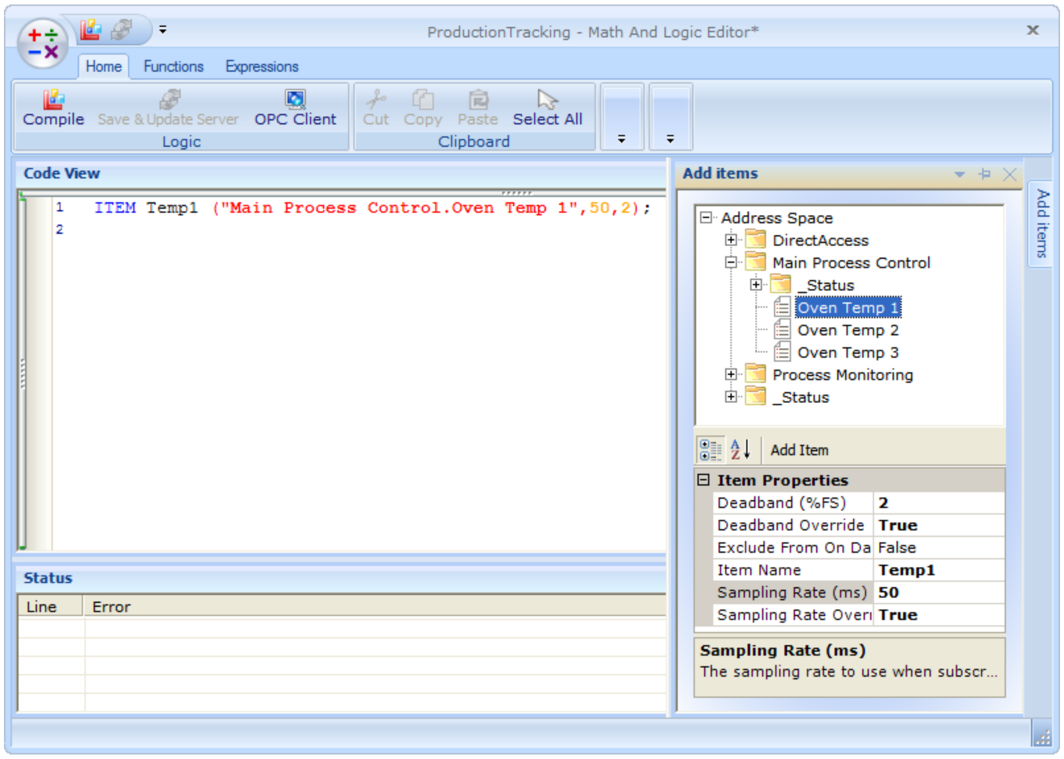
Deadband ('	0
Deadband C	False
Exclude Fro	False
Item Name	ItemName

Item Name
The name assigned to this item in the program.

In this pane, you can browse through all of the configured data items, status items and DirectAccess tags that are available to the OPC server. Select the item you want to use, and then enter a name for it in the ***Item Name*** field. This is the local variable name you will use for the data item in your program.

If you wish, you can override the default deadband and sampling rate for the data item. You can also exclude the item from triggering On Data Change execution. Refer to the Data Item Declarations section for information on how to do this.

Once you have edited the OPC Properties fields as desired, click ***Add Item*** to create a variable declaration in the Code View pane.



```
ITEM Temp1 ("Main Process Control.Oven Temp 1",50,2);
```

Item Properties

Deadband (%FS)	**2**
Deadband Override	**True**
Exclude From On Da	False
Item Name	**Temp1**
Sampling Rate (ms)	**50**
Sampling Rate Overr	**True**

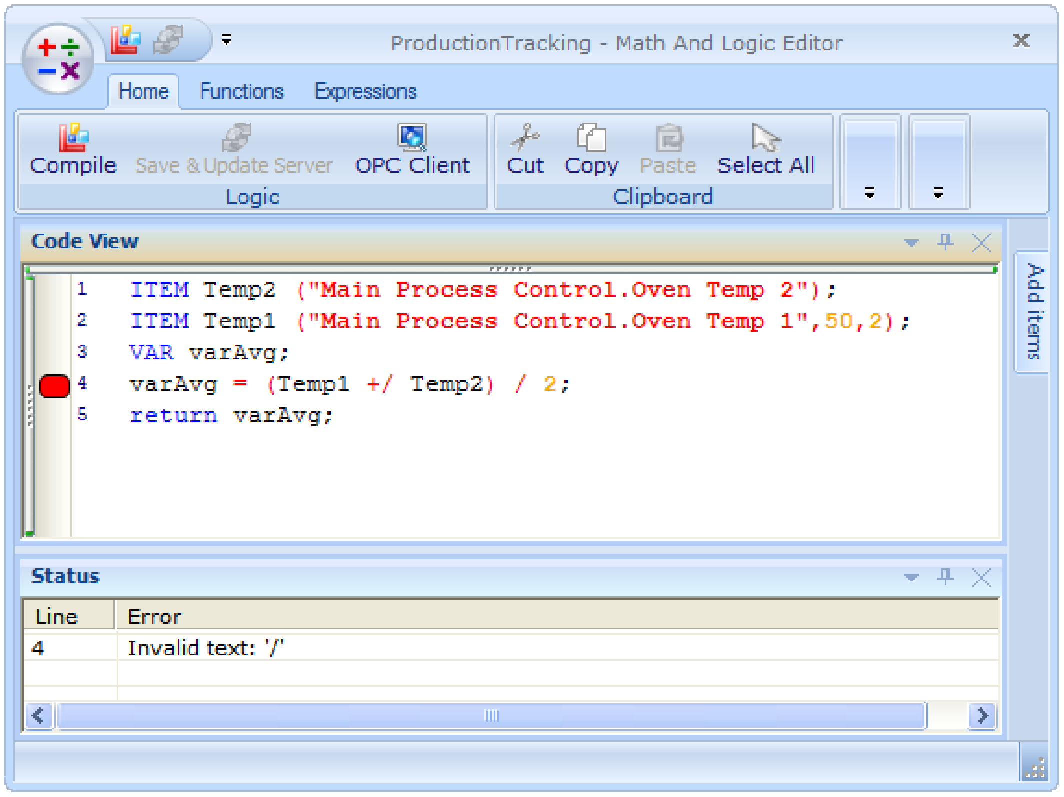
Sampling Rate (ms)
The sampling rate to use when subscr...

The example above shows an ITEM declaration created by the Add Items pane. It includes the Item Name that the user entered, followed by the full ItemID. In this case, the user elected to override the default deadband and sampling rate, so the declaration shows the user-specified values.

Note The ITEM statement is always inserted at the top of the program, regardless of the current cursor position.

Status Pane

The Status pane provides error messages that the compiler generates to help in debugging your program.



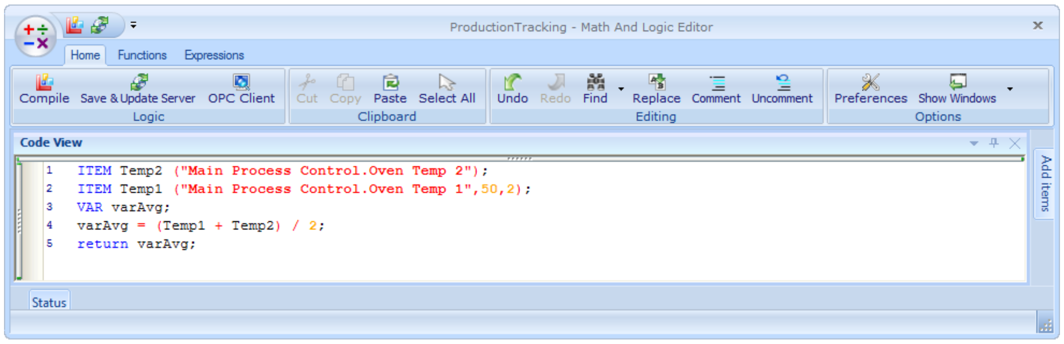
```
1  ITEM Temp2 ("Main Process Control.Oven Temp 2");
2  ITEM Temp1 ("Main Process Control.Oven Temp 1",50,2);
3  VAR varAvg;
4  varAvg = (Temp1 +/ Temp2) / 2;
5  return varAvg;
```

Line	Error
4	Invalid text: '/'

In this example, line 4 has two operators in a row. The Status pane shows that the error occurred in line 4 and gives a description of the problem. The red icon at the left side of the Code View pane visually flags the error location.

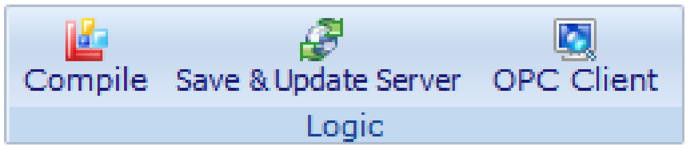
Note The compiler stops after the first error is found. Therefore, if multiple errors are present, only the first one found will be shown.

Home Ribbon



```
1  ITEM Temp2 ("Main Process Control.Oven Temp 2");
2  ITEM Temp1 ("Main Process Control.Oven Temp 1",50,2);
3  VAR varAvg;
4  varAvg = (Temp1 + Temp2) / 2;
5  return varAvg;
```

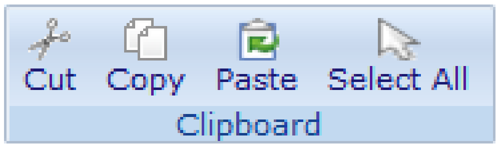
Logic Group



From the Logic group, you can:

- Compile your program. If the compiler detects errors in the code, they will be displayed in the Status pane.
- Save the configuration and update the server. This allows you to run the program without leaving the editor.
- Launch Cyberlogic's OPC client. With it, you can view the input and output data for your program.

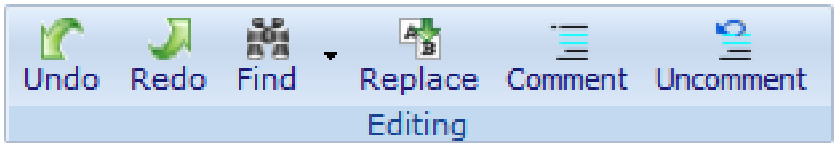
Clipboard Group



This group contains the standard clipboard functions, and the editor also supports their standard keyboard shortcuts:

- Cut (Ctrl-X)
- Copy (Ctrl-C)
- Paste (Ctrl-V)
- Select All (Ctrl-A)

Editing Group



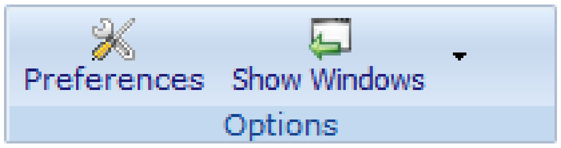
These editing functions, and their keyboard shortcuts, work just as they do in standard text editors:

- Undo (Ctrl-Z)
- Redo (Ctrl-Y)
- Find (Ctrl-F) Click the arrow beside the Find button to access Goto, which allows you to jump directly to any line in the program.
- Replace (Ctrl-H)

To convert one or more lines of code into comments, select them and click the Comment button. It will insert a double-slash at the beginning of each of the selected lines.

To remove the double-slash from these commented lines, select them and click Uncomment.

Options Group

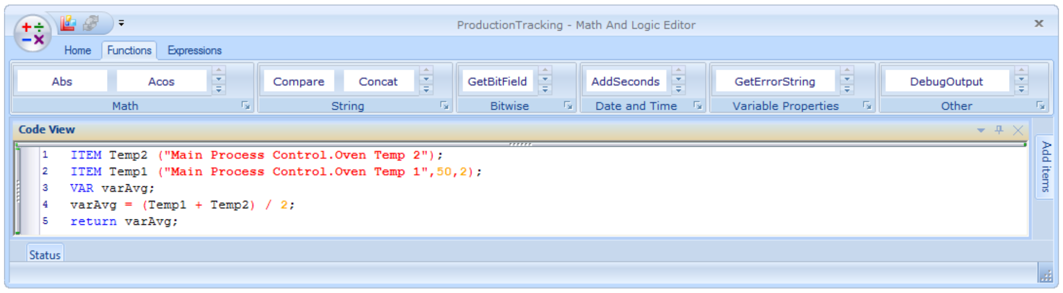


This group lets you adjust the view according to your preferences.

The Preferences button opens an editor that provides an extensive selection of choices that you can use to customize the appearance of the editor. These include font style, size and color, along with indentation options.

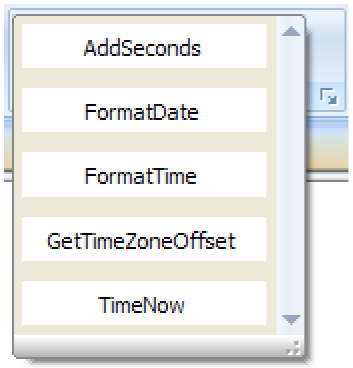
Show Windows lets you choose how to display or hide the Code View, Status and Add Items panes.

Functions Ribbon

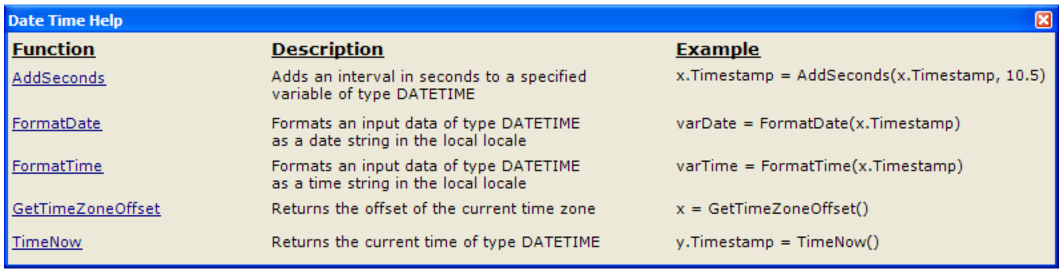


```
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1",50,2);
VAR varAvg;
varAvg = (Temp1 + Temp2) / 2;
return varAvg;
```

Each group on the Functions ribbon contains a gallery of functions and help screens. Click a group's down arrow to expand its gallery of functions.



Here is the gallery for the Date and Time group.



Date Time Help

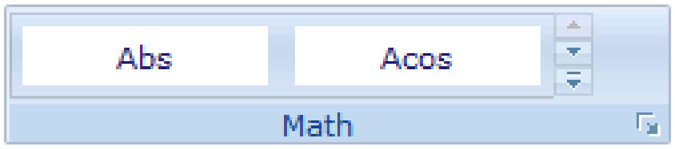
Function	Description	Example
AddSeconds	Adds an interval in seconds to a specified variable of type DATETIME	x.Timestamp = AddSeconds(x.Timestamp, 10.5)
FormatDate	Formats an input data of type DATETIME as a date string in the local locale	varDate = FormatDate(x.Timestamp)
FormatTime	Formats an input data of type DATETIME as a time string in the local locale	varTime = FormatTime(x.Timestamp)
GetTimeZoneOffset	Returns the offset of the current time zone	x = GetTimeZoneOffset()
TimeNow	Returns the current time of type DATETIME	y.Timestamp = TimeNow()

Click the dialog box launcher to open a help screen for the group. This screen contains a list of the functions with a description and example for each. The function names are hyperlinks. Click them to open the help file to the section for that function.

Note On the help screen for the Operators group, the function names are not underlined. This makes it easier to read the individual symbols. However, the names are still hyperlinked to the help file.

Note Another way you can get help with any function, keyword or symbol in the C-logic editor is to place the cursor on the function name or keyword, or select the symbol, and then press <F1>.

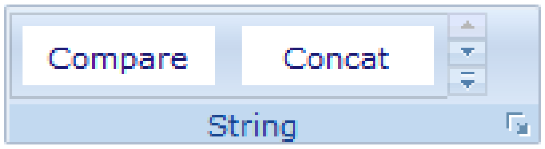
Math Group



This group includes the common trigonometric functions, powers, logarithms, exponentials, and several rounding and conversion functions.

Refer to the Math Functions section for a detailed description of each of these functions.

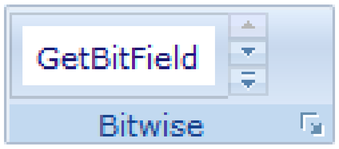
String Group



The String group includes string manipulation, conversion, search, replace and property functions.

Refer to the String Functions section for a detailed description of each of these functions.

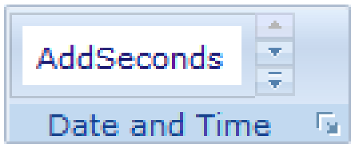
Bitwise Group



The Bitwise functions allow you to extract ranges of bits within a value, and to shift bits left and right.

Refer to the Bitwise Functions section for a detailed description of each of these functions.

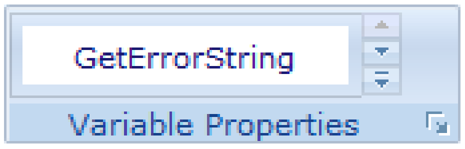
Date and Time Group



The Date and Time Group includes functions to retrieve, format and modify date and time values.

Refer to the Date & Time Functions section for a detailed description of each of these functions.

Variable Properties Group

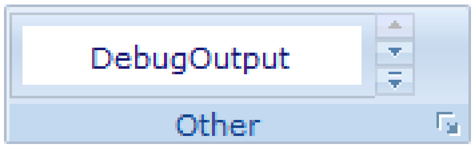


All C-logic variables include Error and Quality properties. This group contains the Error and OPC Quality functions, which simplify the interpretation of these properties.

Refer to the Variable Properties Functions section for a detailed description of each of these functions.

Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

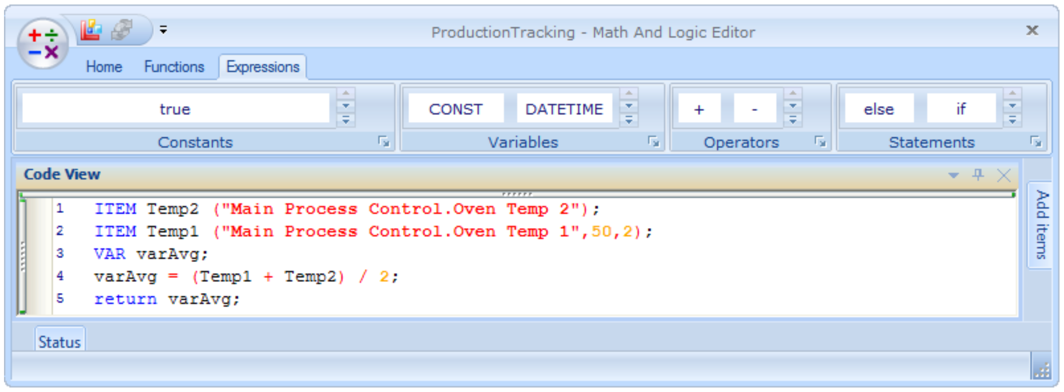
Other Group



These functions help you to debug the program and control its execution.

Refer to the Other Functions section for a detailed description of each of these functions.

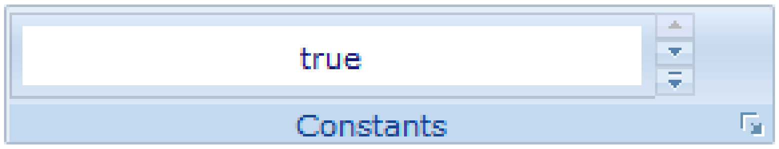
Expressions Ribbon



```
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1",50,2);
VAR varAvg;
varAvg = (Temp1 + Temp2) / 2;
return varAvg;
```

Each group on the Expressions ribbon contains a gallery of elements you can use to create expressions, and corresponding help screens. They are accessed in the same way as the galleries and help screens on the Functions ribbon.

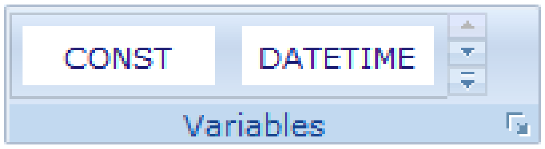
Constants Group



This group contains several pre-defined constant values, including the OPC quality codes, and the mathematical constants π and e.

Refer to the Predefined Constants section for a table of available constants.

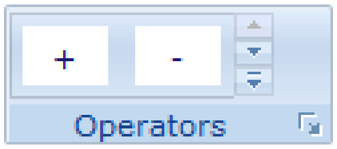
Variables Group



The Variables group allows you to create custom constants, typed and untyped local variables, OPC data item variables, and date and time variables in your program.

Refer to the Local Declarations section for detailed information on the types of constants and variables you can create.

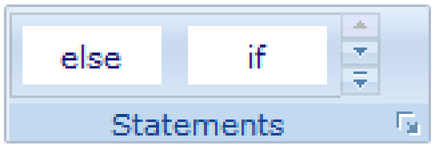
Operators Group



This group provides the operators for mathematical and logical operations.

The Expressions section includes a full list of the available operators.

Statements Group



In this group are the *if/else* and *return* statements.

Refer to the Statements section for details of how to construct these types of statements.

Conversions

The results of your Math & Logic programs may be process values that are not expressed in engineering units. To simplify operations on the data, the Cyberlogic OPC Server allows you to associate a conversion with each data item.

A user can define many different conversions. A number of data items can then use each conversion. As a result, the user need not define the same conversion many times over.

Refer to the Cyberlogic OPC Server Help for a full discussion.

Simulation Signals

The server can simulate the data for each of the data items, including Math & Logic data items, according to a predefined formula. This makes it easy to perform client-side testing without the need for a working Math & Logic program.

A user can define many different types of simulation signals. A number of data items can then use each such signal. As a result, the user need not define the same simulation signal many times over.

The server can generate the following types of simulation signals:

- Read count
- Write count
- Random
- Ramp
- Sine
- Square
- Triangle
- Step

Each signal has parameters that define properties such as amplitude, phase and number of steps.

Refer to the Cyberlogic OPC Server Help for a full discussion.

Alarm Definitions

The Cyberlogic OPC Server supports the OPC Alarms and Events specification, permitting it to generate alarms based on the value of data items.

The user may define many different alarm conditions. A number of data items can then use each such condition. As a result, the user need not define the same alarm condition many times over.

There are two categories of alarms: digital and limit. Digital alarms are normally used with Boolean data items, and limit alarms are normally used with numeric data items. However, both types of alarms may be used with either data type. Alarms cannot be used with string or array data items, or with bit fields larger than 64 bits.

Refer to the Cyberlogic OPC Server Help for a full discussion.

Note Configuring alarms is meaningful only if your client software also supports the OPC Alarms & Events specification. Consult your client software documentation to see what specifications it supports.

Network Connections

Network connections allow you to configure the networks you will use to communicate to network nodes. The network nodes may be PLCs, other OPC servers or other devices that provide data to the OPC server. Typically, the data you receive from these network nodes will be used as inputs or outputs for the Math & Logic programs.

Refer to the driver agent help files for a full discussion.

Database Operations

In addition to providing data to OPC clients in real time, the Cyberlogic OPC Server can store it in a database. The feature that does this is called Data Logger. Once the data is logged, it is available to any application that can access that database. It need not be an OPC client application.

Refer to the Data Logger Help for a full discussion.

OPC Crosslinks

OPC Crosslinks allow you to transfer data from an OPC server or PLC to other OPC servers or PLCs. The data item you read from is called the crosslink input. You may write its value to any number of data items, and these are called crosslink outputs. You can use Math & Logic data items to enable or disable these transfers, and to control when the transfers occur.

Refer to the OPC Crosslink Help for a full discussion.

Saving and Undoing Configuration Changes

The Cyberlogic OPC Server Configuration Editor keeps track of recent configuration changes. Until you save these changes, you can revert to the previously-saved configuration. The editor supports two types of save operations. The standard Save operation saves the changes without updating the server or the connected clients. The Save & Update Server operation saves the changes and also updates the server and all connected clients.

Caution! After you edit the configuration, you must open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** toolbar button, for the changes you have made to take effect. Otherwise, the server will still be running with the old configuration.

Saving Configuration Changes Without Updating the Server

To save the configuration without updating the server, open the ***File*** menu and select ***Save***, or click the ***Save*** button on the toolbar. The changes will be saved but the server will still be running with the old configuration.

Saving Configuration Changes and Updating Server

To save the configuration and update the server, open the ***File*** menu and select ***Save & Update Server***, or click the ***Save & Update Server*** button on the toolbar.

Undoing Configuration Changes

To undo configuration changes and revert to the previously saved configuration, open the ***File*** menu and select ***Undo Changes***, or click the ***Undo Changes*** button on the toolbar.

Configuration Import/Export

The Import/Export feature allows you to export the configuration data to text file format and import configuration data from these exported files. It also allows you to import comma separated values (csv) files from other vendors' OPC servers and programming software.

For details on this important feature and instructions in its operation, refer to the Cyberlogic OPC Server Help.

Editor Options

The Cyberlogic OPC Server Configuration editor has several options that may be set to adjust the operation of the editor to suit your preferences and to set security levels as needed for communication with client software. For a full discussion, refer to the Cyberlogic OPC Server Help.

VALIDATION AND TROUBLESHOOTING

The following sections describe features that will help you to verify and troubleshoot your server's operation. The Data Monitor and Cyberlogic OPC Client allow you to view the data as it is received by the server. Each of those tools allows you to view the Status Items

Data Monitor

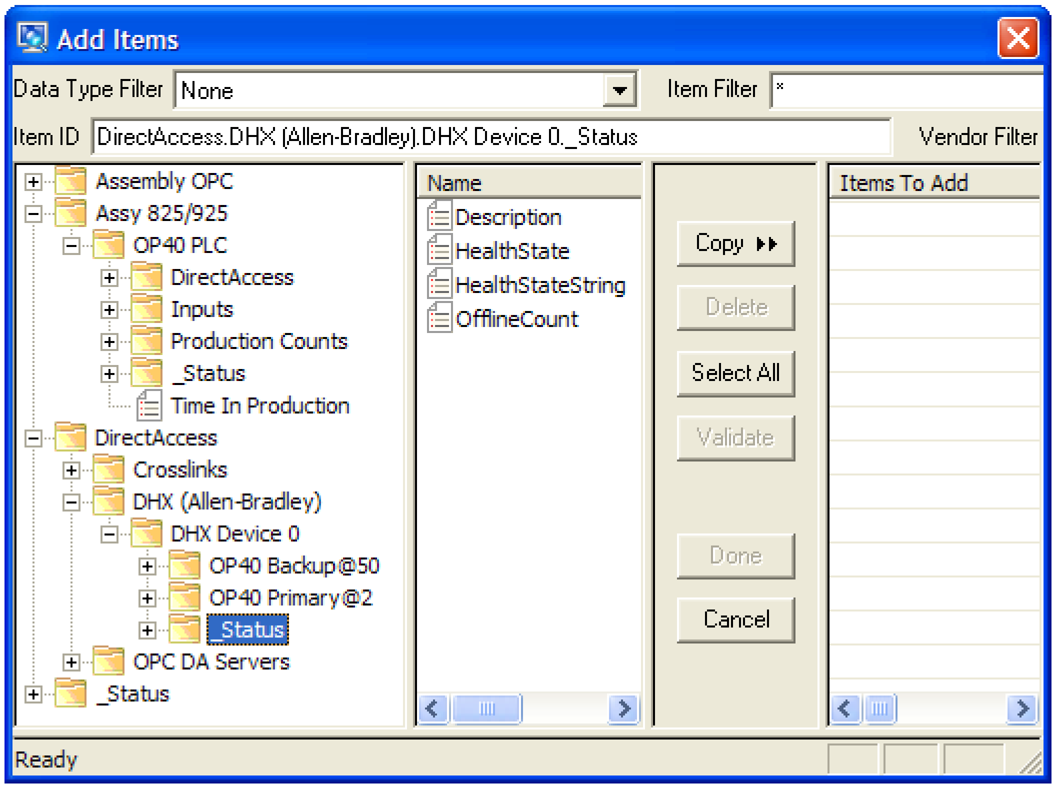
The Data Monitor lets you monitor the values and status of the data items. Its use is described in detail in the Cyberlogic OPC Server Help.

Cyberlogic OPC Client

The Cyberlogic OPC Client is a simple OPC Data Access client that lets you see how the server interacts with a client and lets you test its response to various loads. Its use is described in detail in the Cyberlogic OPC Server Help.

Status Items

The Cyberlogic OPC Server provides status items that are accessible to any connected OPC client application. These items provide health and performance information about the server itself, as well as the network connections, network nodes, devices and crosslinks. For complete information on the standard status items and how to access them, refer to the Cyberlogic OPC Server Help. This section describes status items that are specific to Math & Logic.

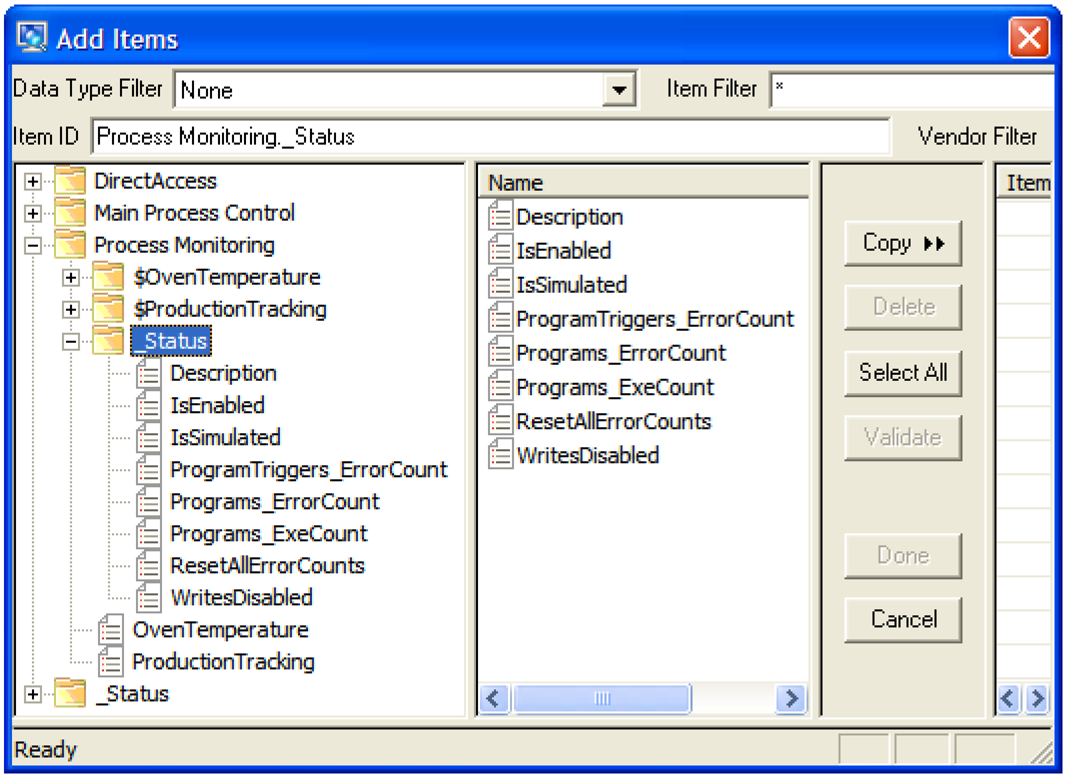


Math & Logic Status Item Definitions

When you connect to the Cyberlogic OPC Server with a client application and browse for items to display, the status items are shown in folders called *_Status*. The contents of each folder depend on the type of item that it provides status for.

- Device Status
- Data Item Status

Device Status



The device status items are in folders called *_Status* located directly under the Math & Logic device branches in the address space. Some device status items are common to all device types, and these are documented in the Cyberlogic OPC Server Help. The status items specific to Math & Logic devices are listed here.

IsEnabled

Indicates whether or not the Math & Logic device is enabled.

The valid values are:

- 0 = Disabled
- 1 = Enabled

Programs_ExeCount

This is the total execution count for all programs associated with this Math & Logic device.

The counter is incremented each time any program associated with this device executes. This counter cannot be reset.

Programs_ErrorCount

This is the total error count for all programs associated with this Math & Logic device.

The counter is incremented each time any program associated with this device generates an error. Refer to ResetAllErrorCounts, above, to see how to reset this counter to zero.

ProgramTriggers_ErrorCount

This is the total error count for all program trigger data items associated with this Math & Logic device. It is incremented each time bad quality data is received for any program trigger data item. Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

EnableDevice_ErrorCount

This is the error count associated with the control input used to enable this Math & Logic device. It is incremented each time bad quality data is received for this data item. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

This item is present only if Use Data Item To Enable is selected on the General Tab of the Math & Logic device.

EnableDevice_ItemID

This is the item ID string associated with the device enable control input.

This item is present only if Use Data Item To Enable is selected on the General Tab of the Math & Logic device.

EnableDevice_UpdateCount

The number of times the device enable control input has changed from enabled to disabled or disabled to enabled. This count is incremented each time IsEnabled changes state.

This item is present only if Use Data Item To Enable is selected on the General Tab of the Math & Logic device.

EnableDevice_LastError

The last error code associated with the device enable control input.

This item is present only if Use Data Item To Enable is selected on the General Tab of the Math & Logic device.

EnableDevice_LastErrorQuality

The last error quality associated with the device enable control input. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

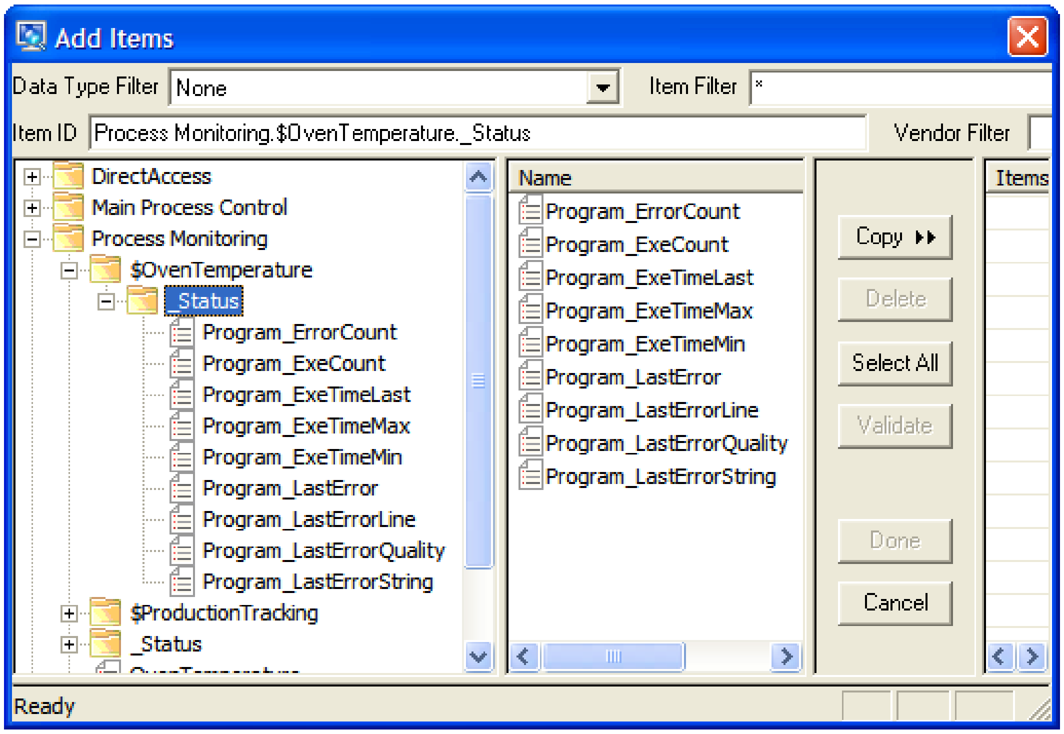
This item is present only if Use Data Item To Enable is selected on the General Tab of the Math & Logic device.

EnableDevice_LastErrorString

The last error code string associated with the device enable control input.

This item is present only if Use Data Item To Enable is selected on the General Tab of the Math & Logic device.

Data Item Status



The folder that contains each Math & Logic data item also contains a folder called *${Item Name}*, where *Item Name* is the name of the Math & Logic data item. Within that folder is a *_Status* folder that contains the status items. Some status items are common to all data item types, and these are documented in the Cyberlogic OPC Server Help. The status items specific to Math & Logic data items are listed here.

Program_ErrorCount

This is the error count associated with the data item's program. It is incremented each time the program execution encounters a runtime error.

Note When a program generates a runtime error, its current execution is terminated.

Program_LastError

This is the last error code associated with this program.

Program_LastErrorLine

This is the line number in the program's source code at which the last runtime error occurred.

Program_LastErrorQuality

This is the last error quality associated with this program. Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

Program_LastErrorString

This is the last error code string associated with this program.

Program_ExeCount

This counter contains the total program execution count. It is incremented each time the program executes.

Program_ExeTimeLast

This is the duration of the last program execution. The time is expressed in microseconds.

Note All programs executing under Windows are subject to interruptions, such as device driver interrupts and preemptions by higher-priority programs. The value of the *Program_ExeTimeLast* includes the time of these interruptions. Therefore, your program's last execution time will generally be lower than the value in the *Program_ExeTimeLast*.

Program_ExeTimeMin

This is the minimum duration of the program's execution. The time is expressed in microseconds.

Note Each ***Save & Update Server*** operation resets the *Program_ExeTimeMin* value to zero.

Program_ExeTimeMax

This is the maximum duration of the program's execution. The time is expressed in microseconds.

Note All programs executing under Windows are subject to interruptions, such as device driver interrupts and preemptions by higher-priority programs. The value of the *Program_ExeTimeMax* includes the time of these interruptions. Therefore, your program's maximum execution time will generally be lower than the value in the *Program_ExeTimeMax*.

Note Each ***Save & Update Server*** operation resets the *Program_ExeTimeMax* value to zero.

ProgramTrigger_ErrorCount

This counter contains the total error count associated with the trigger program execution control input. It is incremented each time bad quality data is received for this input. Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

ProgramTrigger_ItemID

This is the Item ID string associated with the trigger program execution control input.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

ProgramTrigger_LastError

This is the last error code associated with the trigger program execution control input.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

ProgramTrigger_LastErrorQuality

This is the last error quality associated with the trigger program execution control input. Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

ProgramTrigger_LastErrorString

This is the last error code string associated with the trigger program execution control input.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

ProgramTrigger_LastValue

This is the last data value associated with the trigger program execution control input.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

ProgramTrigger_UpdateCount

This counter contains the total update count for the trigger program execution control input. This counter is incremented each time an OnDataChange callback delivers a new value for the trigger program execution control input.

This item is present only if a data item is configured to trigger the program execution. This may be done on the Settings Tab of the Math & Logic device or in the Advanced... settings for the Math & Logic data item.

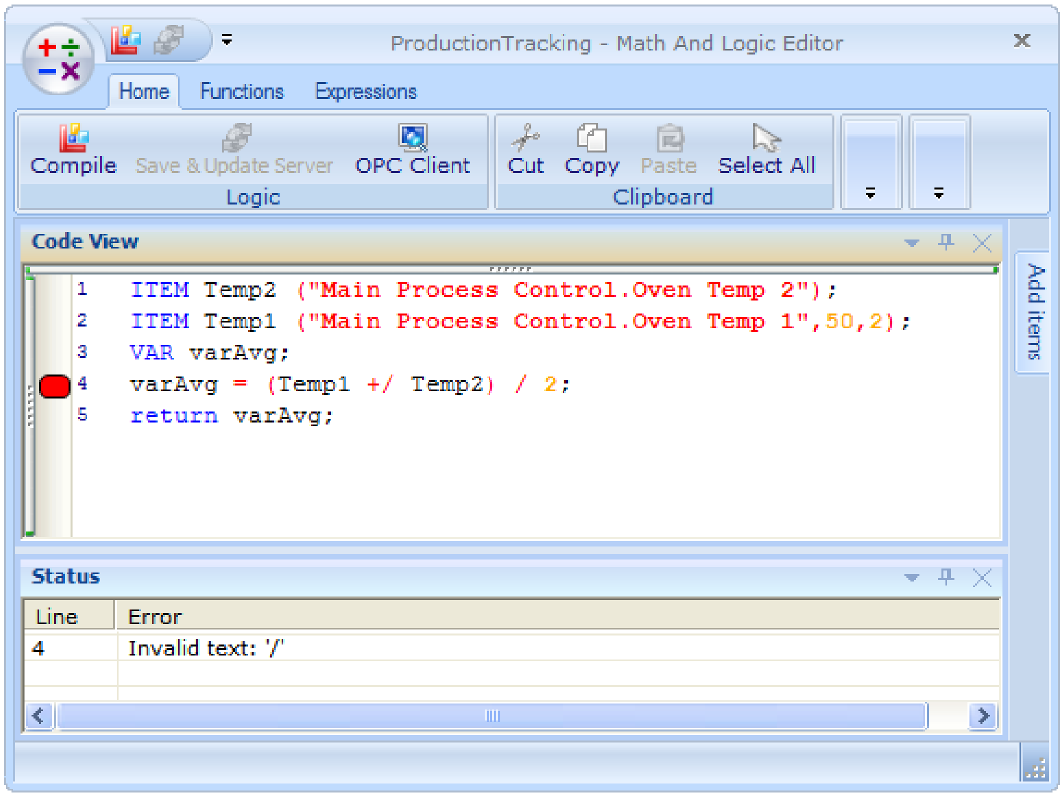
Debugging Aids

Once you have written it, your program may not work as expected. It may fail to compile because of fatal syntax errors, it may compile successfully but not run, or it may run with incorrect results.

To help you fix these problems, the Cyberlogic OPC server includes a number of debugging tools.

Status Window

If the compiler detects errors in your code, it displays an explanation of these errors in the status window.

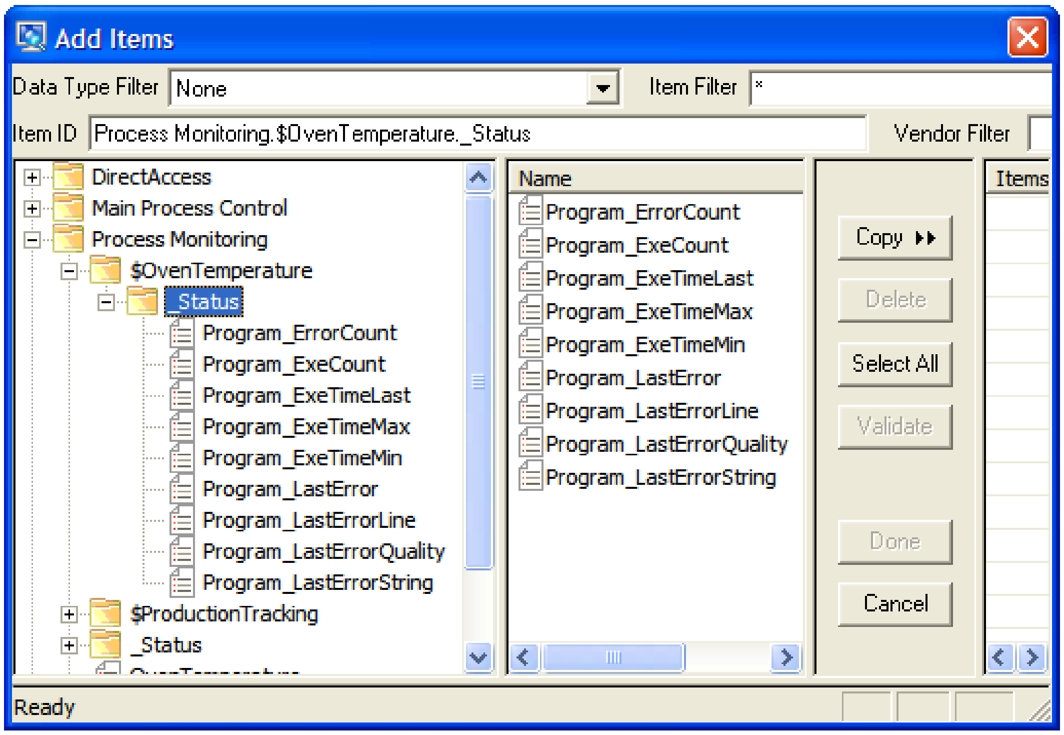


```
ITEM Temp2 ("Main Process Control.Oven Temp 2");
ITEM Temp1 ("Main Process Control.Oven Temp 1",50,2);
VAR varAvg;
varAvg = (Temp1 +/ Temp2) / 2;
return varAvg;
```

Line	Error
4	Invalid text: '/'

In this example, line 4 has two operators in a row. The Status pane shows that the error occurred in line 4 and gives a description of the problem. The red icon at the left side of the Code View pane visually flags the error location.

Status Items



The status items are data items, available to any OPC client application, that provide key information about the functioning of your program. You can find detailed information about them in the Status Items section.

You can use the following procedure to evaluate your program at runtime:

1. Examine the Program_ExeCount status item to verify that the program is running. This count will increment each time the program executes. If it is not increasing, the program is not running.

2. Examine the Program_ErrorCount to see if any errors have been recorded. This data item records the number of runtime errors that the software has detected.

Note The error counts for all programs can be reset to zero by an Off to On transition of the ResetAllErrorCounts status item. This item is located in the *_Status* folder directly under the Math & Logic device.

3. If the Program_ErrorCount is not 0, then check the Program_LastErrorString. This string provides a description of the last detected runtime error.

DebugOutput Function

It is frequently helpful to be able to see the values of internal variables as the program is running. The DebugOutput function allows you to do this. With it, you can output any value in the program to a data item that is available for viewing in an OPC client application.

Writing to debug outputs extends the execution time. Therefore, you should remove or comment out all DebugOutput calls once the program is running as desired.

Public Variables

You can declare typed variables in your program to be public. These Public Variables will then be available to the OPC clients, appearing just as regular, configured OPC data items. This makes it very easy for you to view these variables during program execution.

It is important to understand, however, that writing to public variables is much slower than writing to local variables. If you make a variable public for debugging, you should change it back to a local variable after the program is running as desired.

APPENDIX A: C-LOGIC LANGUAGE REFERENCE

You use the C-logic programming language to write the Math & Logic programs. Its syntax is similar to the C programming language, as you can see in the following sample program. The program demonstrates the use of local variables, comments, "if" statements, function calls, and simple arithmetic expressions. It also demonstrates how the result is returned.

Note To provide you with additional examples, the software also includes a configuration file with a set of sample programs. These will help you to understand how the various functions work. They will also give you ideas about what you can do with C-logic, and they can be modified and used in creating your own programs. You will find a listing of some of the sample programs in Appendix D: Sample Programs.

This program adds the absolute values of three data items, excluding those that have a bad quality. The local variable declarations are at the top of the program, followed by the executable statements.

```
ITEM x ("Device.Folder.ItemX");        // Item ID, default sampling & deadband
ITEM y ("Device.Folder.ItemY",50,10);  // Item ID, sampling rate, deadband
ITEM z ("Device.Folder.ItemZ",,10.0);  // Item ID, default sampling, deadband
VAR varX;                              // Local variable for X
VAR varY;                              // Local variable for Y
VAR varZ;                              // Local variable for Z
VAR varResult;                         // Result

varX = 0;
varY = 0;
varZ = 0;


if(IsQualityBAD(x) && IsQualityBAD(y) && IsQualityBAD(z))
{
       varResult.Quality = QUALITY_SENSOR_FAILURE;
}
else
{
       if(!IsQualityBAD(x))
       {
              varX = abs(x);
       }
       if(!IsQualityBAD(y))
       {
              varY = abs(y);
       }
       if(!IsQualityBAD(z))
       {
              varZ = abs(z);
       }
       varResult = varX + varY + varZ;
}
return varResult;
```

General Rules

You must place all local declarations for constants and variables at the beginning of the program, before any executable statements.

All names and keywords in the language are case-insensitive. Numeric values that contain letters (0t47, 0x5FC2, -2.5e-4) are also case-insensitive. However, character and string values ('A', "Crosslink") are case-sensitive.

At the start of each program execution, the values of all public variables and variables of type ITEM are set to the current values of their associated data items. If the OPC server receives new data from an external device, the OPC data items' values will be updated, but the new values will not be applied to the corresponding local variables until the beginning of the next program execution. For more information on data item variables, refer to the Data Item Declarations section.

Variables used in expressions must have GOOD or UNCERTAIN quality, or the expression cannot be evaluated. For the result to have GOOD quality, all operands must have GOOD quality. If any operands have UNCERTAIN quality, the result will have UNCERTAIN quality. An attempt to operate on a variable with BAD quality will terminate the processing of the expression and generate an error.

If a statement, function or expression generates an error, the current execution of the program is terminated. The error count is incremented, and the error information is reported through the program's Data Item Status items.

A conversion between signed and unsigned integers of different lengths generates an overflow error if the requested type cannot hold the value. For example:

- A signed 16-bit integer value of –1 will overflow if you try to convert it to an unsigned 8-bit integer.
- An unsigned 16-bit integer value of 255 will overflow if you try to convert it to a signed 8-bit integer.
- The same rules apply to conversions between any pair of integers of 8, 16, 32 or 64 bits, when the lengths are different.

However, for conversions between signed and unsigned integers with the same number of bits, the value is *not checked* for overflow. For example:

- A signed 8-bit integer value of –1 will convert to an unsigned 8-bit integer value of 255.
- An unsigned 8-bit integer value of 254 will convert to a signed 8-bit integer value of –2.
- The same rules apply to conversions between any pair of integers of 8, 16, 32 or 64 bits, when the lengths are the same.

Comments

The C-logic compiler supports single-line comments. Comments begin with two forward slashes (//) and are terminated by the next new line character. They cannot extend to a second line.

```
// This is a valid comment
```

Comments do not affect the generated code or the program's execution speed, but greatly improve readability.

Constants

There are three types of constants:

- Integer Constants (decimal, hex, octal, binary, character)
- Floating Point Constants
- String Constants

Several commonly-used constants are predefined and need not be declared. You will find a list of these in the Predefined Constants section.

The Linear Conversion Sample Program and Two Sines Sample Program include examples of the use of constants.

Integer Constants

Integer constants can be of type decimal, hexadecimal, octal or binary. Decimal numbers must not contain a decimal point.

The format is: [whitespace] [{+ | –}] [0 { x | X | t | T| b | B}] [digits]

Binary, octal and hexadecimal constants are always taken as a 64-bit signed integer, with the most significant bit extended to fill any unspecified bit positions. Therefore, if the value you specify has 1 as the most significant bit, it will be interpreted as a negative number. To force such a value to be interpreted as positive, you must add a leading 0.

For example, 0xFF will be taken as -1. Its MSB is 1, and that will be extended to produce the 64-bit signed integer FFFFFFFFFFFFFFFF. If you intend it to be taken as 255, you must enter the value as 0x0FF. Doing so forces the MSB to be 0, and the 64-bit representation will then be 00000000000000FF.

This is a concern for binary numbers with a most significant bit of 1, octal numbers with a most significant digit in the range 4-7, and hexadecimal numbers with a most significant digit in the range of 8-F.

This is not an issue for decimal numbers, because they are always assumed to be positive unless the "–" sign is present.

Here are some examples:

```
100                 // Decimal number
0x12                // Hexadecimal number (18)
0x82                // Hexadecimal number (-126)
0x082               // Hexadecimal number (130)
0t12                // Octal number (10)
0t72                // Octal number (-6)
```

```
0t072             // Octal number (58)
0b101             // Binary number (-3)
0b0101            // Binary number (5)
```

Character Constants

Character constants are also integer constants. They consist of a character enclosed in single quotation marks. For example:

```
'A'               // ASCII A
' '               // Space character
```

Escape Sequences

Escape sequences are character combinations that consist of a backslash (\) followed by a letter or by a combination of digits. They provide literal representations of carriage control codes, nonprinting characters, and other characters that have special meanings. If you want to create a character constant that contains one of these special characters, you must use an escape sequence. An escape sequence is regarded as a single character, and is therefore valid as an integer constant.

The following table lists the ANSI escape sequences supported by C-logic and the characters they represent.

ANSI Escape Sequences	
Escape Sequences	**Represented Character**
\a	Bell (alert)
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\x*hhhh*	Unicode character in hexadecimal notation.

If a backslash precedes a character that does not appear in the table, the character is undefined. The compiler handles this undefined character as the character itself. For example, \c is treated as c.

Some escape sequence examples:

```
x = '\n';         // Set x to new line character
```

```
x = '\x221e';      // Set x to Unicode 0x221e (infinity sign)
```

Floating Point Constants

A floating-point constant is a decimal number that represents a signed real number. It includes an integer portion, a fractional portion, and an exponent.

You can omit either the digits before the decimal point (the integer portion), or the digits after the decimal point (the fractional portion), but not both. You can omit the decimal point only if you include an exponent. No white-space characters are permitted. Here are some examples:

```
100.                 // = 100.0
100.25               // = 100.25
1.575E1              // = 15.75
1575e-2              // = 15.75
-2.5e-3              // = -0.0025
25E-4                // =  0.0025

100                  // INVALID: no decimal point and no exponent
-2.5 e-3             // INVALID: contains a space
```

String Constants

A string constant is a sequence of characters enclosed in double quotes ("). As part of the string, you may want to include one of the special characters listed in the ANSI Escape Sequences table. To do this, simply include its escape sequence in the string. Here are some examples:

```
"Hello"                         // String text
"123"                           // Numeric string
"I use \"double quotes\""       // String with double quotes
"I use \\ single backslash"     // String with a backslash
"A one in Chinese is \x4e00"    // String with a Unicode character
```

You can use numeric string constants in arithmetic expressions. They will be converted to numbers before the expression is evaluated. For example:

```
x = y + "123";                  // This is a legal expression
x = y + "123abc";               // This is an illegal expression
```

Predefined Constants

Several commonly-used constants are predefined, and need not be declared. Here is a list of these constants:

Constant Name	**Value**	**Description**
true	-1	Boolean true
false	0	Boolean false
pi	3.14159... (39 digits)	Math constant pi
e	2.71828... (39 digits)	Math constant e
QUALITY_GOOD	0xC0	GOOD quality, non-specific
QUALITY_LOCAL_OVERRIDE	0x0D8	GOOD quality, local override
QUALITY_BAD	0x00	BAD quality, non-specific
QUALITY_CONFIG_ERROR	0x04	BAD quality, configuration error
QUALITY_NOT_CONNECTED	0x08	BAD quality, not connected
QUALITY_DEVICE_FAILURE	0x0c	BAD quality, device failure
QUALITY_SENSOR_FAILURE	0x10	BAD quality, sensor failure
QUALITY_LAST_KNOWN	0x14	BAD quality, last known value
QUALITY_COMM_FAILURE	0x18	BAD quality, communication failure
QUALITY_OUT_OF_SERVICE	0x1C	BAD quality, out of service
QUALITY_WAITING_FOR_INITIAL_DATA	0x20	BAD quality, waiting for initial data
QUALITY_UNCERTAIN	0x40	UNCERTAIN quality, non-specific
QUALITY_LAST_USABLE	0x44	UNCERTAIN quality, last usable value
QUALITY_SENSOR_CAL	0x50	UNCERTAIN quality, sensor not accurate
QUALITY_EGU_EXCEEDED	0x54	UNCERTAIN quality, engineering units exceeded
QUALITY_SUB_NORMAL	0x58	UNCERTAIN quality, sub-normal
QUALITY_LIMIT_OK	0x00	Not limited
QUALITY_LIMIT_LOW	0x01	Low limited
QUALITY_LIMIT_HIGH	0x02	High limited
QUALITY_LIMIT_CONST	0x03	Constant
S_OK	0x00000000	SUCCESS OK
S_FALSE	0x00000001	SUCCESS Some failures occurred
E_FAIL	0x80004005	ERROR The operation failed
E_INVALIDARG	0x80070057	ERROR Invalid Argument
E_ACCESSDENIED	0x80070005	ERROR Not Authorized

Local Declarations

You must place all local declarations at the beginning of the program, before any executable statements. There are four types of local declarations:

- Constant Declarations
- Data Item Declarations
- Typed, Untyped and Public Variable Declarations
- Date & Time Variable Declarations

Each variable has a set of Standard Variable Properties that you can access within your program. The Date & Time Variables have additional properties related to date and time, as explained in the Date & Time Variable Declarations section.

Constant Declarations

A constant declaration assigns a name to a constant. You can then use the name in your program in place of the actual constant value. Here is the syntax:

CONST *Name* **(** *Value***);**

Where:

Name	Unique name assigned to this constant.
Value	Constant value.

Examples:

```
CONST ItemId ("Device.Folder.ItemX");   // String constant
CONST BitMask (0x03);                   // Integer constant
CONST Multiplier (100.5);               // Floating point constant
```

Data Item Declarations

An item declaration assigns a variable name to an OPC data item. This declaration is required before you can use the value and properties of that data item within your program.

Caution! At the start of each program execution, the values of all public variables and variables of type ITEM are set to the current values of their associated data items. If the OPC server receives new data from an external device, the OPC data items' values will be updated, but the new values will not be applied to the corresponding local variables until the beginning of the next program execution.

Item declarations use the following syntax:

ITEM *ItemName* **(***ItemID*[,*SamplingInterval*[,*Deadband*[,*ExcludeFromOnDataChangeTrigger*]]]**);**

Where:

ItemName	Unique variable name assigned to this item.
ItemID	Item ID string for the OPC data item. This can either be a full or relative Item ID. See below for more information on relative Item IDs.
SamplingInterval	Optional sampling interval in milliseconds. The valid range is 20-4000000000. When specified, it overrides the default sampling interval set at the device level. If not used, it must be left blank.
Deadband	Optional deadband in percent of full scale. The valid range is 0.0 to 100.0. When specified, it overrides the default deadband set at the device level. If not used, it must be left blank.
ExcludeFrom OnDataChangeTrigger	Optional flag. When true, it excludes this data item from triggering On Data Change executions of a program. Commonly used when declaring data items that are used only as outputs. If omitted, it defaults to false.

Examples:

```
ITEM x ("Device.Folder.ItemX");       // Item ID, default sampling & deadband
ITEM x ("Device.Folder.ItemY",50);    // Item ID, sampling, default deadband
ITEM x ("Device.Folder.ItemY",50,10); // Item ID, sampling & deadband
ITEM x ("Device.Folder.ItemZ",,10.5); // Item ID, default sampling, deadband
ITEM x ("Device.Folder.Out1",,,true); // Exclude from On Data Change trigger
```

In addition to the full Item IDs, the *ItemID* string can specify a data item reference that is relative to the data item being programmed. Here are the rules for creating relative Item IDs:

- An empty string ("") identifies the current data item.
- Each period ('.') in front of the Item ID represents one level back from the current data item.

Examples:

```
ITEM x ("Device.Folder.Program");     // Current program's Item ID
ITEM x ("");                          // Same as above
ITEM x (".ItemX");                    // Same as "Device.Folder.ItemX"
ITEM x ("..Folder2.ItemY");           // Same as "Device.Folder2.ItemY"
```

You cannot write to individual properties of variables of type ITEM. You can modify them only by assigning a new value to the entire item variable.

Example:

```
ITEM x ("ItemId");
x.Quality = QUALITY_SENSOR_FAILURE;   // Invalid
x = y;                                // Valid (y.Quality may be BAD)
```

The ability to write to ITEM variables is one way you can create programs with multiple outputs; another way is to use Public Variables. An assignment to an ITEM variable updates the server's cache for the associated data item. This type of assignment persists between program executions.

Caution!

If a variable of type ITEM is associated with a bit or register in an external device, the assignment will not initiate a write to that device. Only the cached value in the server will be updated.

Furthermore, it is likely that the assigned data will be overwritten when the server obtains new data from the external device. Therefore, you should assign values to data items only if they are simulated with the Echo simulation selection.

If you need to write to an external device, setup a crosslink that is triggered from your program. Refer to the OPC Crosslink Help for more information.

Caution!

Assignments to ITEM variables are much slower than similar assignments to the local variables. Avoid multiple assignments to the same variable during the same program execution.

Typed, Untyped and Public Variable Declarations

A variable declaration creates a local or public variable and assigns it a name. This declaration is required before you can use the variable within your program.

Local variables may be typed or untyped. When the program makes an assignment to a typed variable, it converts the value to the variable's type. When it makes an assignment to an untyped variable, the variable's value assumes the type of the assigned expression.

Public variables are a form of typed variable with the added capability of being browsed and read just like a configured OPC server data item.

Arrays of typed local and public variables are also supported.

Local and public variables support the Standard Variable Properties. The *Timestamp* property of all local variables is initialized to the current time at the start of each program execution. This value will change when the program makes assignments to a variable or to its *Timestamp* property. The *Timestamp* of public variables, however, is modified only during the variable assignments, and individual writes to the *Timestamp* property (as well as Quality and Error) are not allowed.

Untyped Variables

Here is the syntax for an untyped local variable declaration:

VAR *Name* [**(***Value***)**]**;**

Where:

Name	Unique name assigned to this variable.
Value	Optional initialization value. If not used, the variable will default to BAD quality and no data value.

Example:

```
VAR varX;                // Uninitialized untyped variable
var x(5);                // Untyped variable initialized to 5
```

Untyped variables support the Standard Variable Properties. The Quality and Error properties of uninitialized untyped variables are set as follows:

```
Quality = QUALITY_BAD;      // BAD quality
Error = 0x80004005;         // Failure code E_FAIL
```

Initialized untyped variables are set to their initialization value only once. This is done when the variable is created, which happens before the first program execution. Subsequent changes to the variable's value are persistent across program executions. Therefore, each program execution starts with variables whose values are the same as they were at the end of the previous program execution.

Caution! Data assigned to typed and untyped variables persists between program executions, but does not persist following the Save & Update Server operation. All local variables are reinitialized each time a Save & Update Server is done.

Use public variables if it is crucial that your variables persist across the Save & Update Server operations.

Typed Variables

Here is the syntax for the typed local variable declarations:

VarType Name [**(** *Value* **)**]**;**

*VarType Name***[** *NumberOfElements*[,*LowerBound*]**]** [**({**Value, ...**})**]**;**   // Array

Where:

Name	Unique name assigned to this variable.
Value	Optional initialization value(s). If not used for all elements of an array, the unused positions must be left blank, and those elements will initialize to the default value for their data type.
NumberOfElements	Number of elements in the array.
LowerBound	Optional lower bound value for the array. If not used, it defaults to 0.
VarType	Variable type that can be one of the following:

Note A Variable Type entry in the following table may contain multiple names. Each of these names are equivalent and can be used interchangeably.

Variable Type	Description
bool, boolean, VT_BOOL	Binary boolean (true or false)
string, VT_BSTR	String
sbyte, VT_I1	Signed 8-bit integer
int16, VT_I2	Signed 16-bit integer
int32, VT_I4	Signed 32-bit integer
int, int64, VT_I8	Signed 64-bit integer
byte, VT_UI1	Unsigned 8-bit integer
uint16, VT_UI2	Unsigned 16-bit integer
uint32, VT_UI4	Unsigned 32-bit integer
uint64, VT_UI8	Unsigned 64-bit integer
float, VT_R4	IEEE 32-bit floating point number
double, VT_R8	IEEE 64-bit floating point number
VT_CY	Currency
VT_DATE	Date

Examples:

```
int nNumber;                // Signed 64-bit integer variable
int x(5);                   // Signed 64-bit integer initialized to 5
double nNumber;             // IEEE 64-bit floating point variable
bool bFlag;                 // Boolean flag
VT_BOOL bFlag;              // Boolean flag
VT_R8 aArray[10];           // Array of type VT_R8 with lower bound = 0
VT_R8 aArray[10,1];         // Array of type VT_R8 with lower bound = 1
VT_R8 aArray[2] ({1,2});    // Array of type VT_R8 initialized to 1 and 2
```

Typed variables support the Standard Variable Properties. The Quality and Error properties of uninitialized typed variables are set as follows:

```
Quality = QUALITY_BAD;      // BAD quality
Error = 0x80004005;         // Failure code E_FAIL
```

Initialized typed variables are set to their initialization value(s) only once. This is done when the variable is created, which happens before the first program execution. Subsequent changes to the variable's value are persistent across program executions. Therefore, each program execution will start with variables whose values are the same as those at the end of the previous program execution.

Caution! Data assigned to typed and untyped variables persists between program executions, but does not persist following the Save & Update Server operation. All local variables are reinitialized each time a Save & Update Server is done.

Use public variables if it is crucial that your variables persist across the Save & Update Server operations.

Public Variables

Public variables are a form of typed variable with the added capability of being browsed and read just like any configured OPC server data item.

Unlike the ITEM variables, which refer to data items that already exist in the server's address space, public variables need not be configured outside of a program. They are automatically added to the folder named *${Item Name}*, which is located in the same folder as the associated data item.

Public variables are convenient for defining inputs and outputs for a program. They can also help you debug a program by temporarily making all or some of the program's variables public. By doing so, internal variables are made externally visible, so that you can monitor them during the program execution.

Caution! At the start of each program execution, the values of all public variables and variables of type ITEM are set to the current values of their associated data items. If the OPC server receives new data from an external device, the OPC data items' values will be updated, but the new values will not be applied to the corresponding local variables until the beginning of the next program execution.

Caution! Assignments to public variables are much slower than similar assignments to local variables. Therefore, you should not declare all variables public, except when debugging.

Here is the syntax for the public variable declarations:

public *VarType Name*
    [**(***Value*[,*DisableWrites* [,*SamplingInterval*[,*Deadband*[,*ExcludeFromOnDataChangeTrigger*]]]]**)**];

public *VarType Name***[***Elements*[,*LowerBound*]**]**
    [**({** *Value*,...**}**[,*DisableWrites* [,*SamplingInterval*[,*Deadband*[,*ExcludeFromOnDataChangeTrigger*]]]]**)**];

VarType	Variable type that is specified in the same manner as for typed variables.
Name	Unique name assigned to this variable.
Value	Optional initialization value(s). If not used for all elements of an array, the unused positions must be left blank, and those elements will initialize to the default value for their data type.

DisableWrites	Boolean flag indicating whether external writing to the variable should be disabled. Note, however, that assignments inside a program are always enabled. Defaults to false if not specified.
SamplingInterval	Optional sampling interval in milliseconds. The valid range is 20-4000000000. When specified, it overrides the default sampling interval set at the device level. If not used, it must be left blank.
Deadband	Optional deadband in percent of full scale. The valid range is 0.0 to 100.0. When specified, it overrides the default deadband set at the device level. If not used, it must be left blank.
ExcludeFrom OnDataChangeTrigger	Optional flag. When true, it excludes this data item from triggering On Data Change executions of a program. Commonly used when declaring data items that are used only as outputs. If omitted, it defaults to false.
NumberOfElements	Number of elements in the array.
LowerBound	Optional lower bound value for the array. If not used, it defaults to 0.

Examples:

```
public int x;                  // Uninitialized, all defaults
public int x(5);               // Initialized, rest default
public double x[10]({1,2,3}); // Initialized public array
public int x(5,,100);          // Initialized, sample @ 100 ms

public double x[10]({1,2,3},true, 100,,true);
       // Partly-initialized array, disable writes,
       // sample at 100 ms, exclude from ODC triggering
```

Unlike untyped and typed local variables, uninitialized public variables initialize to the default value for their data type: zero for numeric types, and an empty string for strings.

Initialized public variables are set to their initialization values only once. This is done when the variable is created, which happens before the first program execution. Subsequent changes to the variable's value are persistent across program executions. Therefore, each program execution will start with variables whose values are the same as those at the end of the previous program execution, unless they were written to externally.

Note Data assigned to public variables persists between program executions and Save & Update Server operations. This is different than the typed and untyped local variables, which are reinitialized each time a Save & Update Server is done.

Public variables support the Standard Variable Properties. Similar to the ITEM variables, you cannot write to the individual properties of public variables. You can modify them only by assigning a new value to the entire variable.

Example:

```
public int x;
x.Quality = QUALITY_SENSOR_FAILURE;      // Invalid
x = y;                                   // Valid (y.Quality may be BAD)
```

Date & Time Variable Declarations

A date and time variable declaration creates a variable of type DATETIME. This type of variable simplifies date and time manipulations. Date and time declarations use the following syntax.

DATETIME *Name*[**(***Type***)**]**;**

Where:

Name	Unique name assigned to this variable.
Type	Optional time zone type. Valid values are UTC (Coordinated Universal Time) or Local. If not specified, it defaults to Local.

Variables of type DATETIME have the Standard Variable Properties, and the following additional properties. These are interpreted as either Local or UTC time, according to the Type setting. They can be read and written individually. The initial setting for DATETIME variables is January 1, 2010 at 00:00:00.000

Year	Valid range is 1601 to 30827

Month	January = 1
	February = 2
	March = 3
	April = 4
	May = 5
	June = 6
	July = 7
	August = 8
	September = 9
	October = 10
	November = 11
	December = 12

DayOfWeek	Sunday = 0
	Monday = 1
	Tuesday = 2
	Wednesday = 3
	Thursday = 4
	Friday = 5
	Saturday = 6

Day	Day of the month. Valid range is 1 to 31.
Hour	Valid range is 0 to 23

Minute	Valid range is 0 to 59
Second	Valid range is 0 to 59
Milliseconds	Valid range is 0 to 999

You can access DATETIME variables directly, in which case the value is an OPC-compatible timestamp. An OPC timestamp is a 64-bit signed integer (int64) value equal to the number of 100-nanosecond intervals since January 1, 1601. It is always expressed in UTC time.

Example:

```
// Time in seconds since 12:00 am, Jan 30, 2009
ITEM x ("Device.Folder.TimeInSeconds");
VAR varString;
DATETIME dt;

// Set start time to 12:00 am, Jan 30, 2009
dt.Year = 2009;
dt.Month = 1;
dt.Day = 30;
dt.Hour = 0;
dt.Minute = 0;
dt.Second = 0;
dt.Milliseconds = 0;

// Add seconds since the start time
dt = AddSeconds(dt, x);

// Format the output string
varString = Format("Reported time & date: %d/%d/%d %d:%d:%d\n",
dt.Month, dt.Day, dt.Year, dt.Hour, dt.Minute, dt.Second);
return varString;
```

The Array to Date & Time Sample Program and Time in Your Time Zone #1 Sample Program also include examples of the use of date and time declarations and properties.

Standard Variable Properties

All variable types have the following standard properties.

Value	Data value
Quality	Variable's quality
Timestamp	Variable's timestamp of type DATETIME
Error	Variable's error code

Examples:

```
x.Value           // Value of x (same as x)
```

```
x.Quality          // Quality of x
x.Timestamp        // Timestamp of x (as 64-bit number)
x.Error            // Error code of x
```

Properties of local variables can be read and written individually. ITEM and public variable properties can be read, but cannot be written. When reading a property, the returned quality is always GOOD, and the error code is set to zero (S_OK).

The Maintenance Time Tracking Sample Program and Linear Conversion Sample Program include examples of how to test and set variable properties.

Expressions

The C-logic language supports four types of expressions:

- Arithmetic
- Relational
- Logical
- Bitwise

The evaluation of expressions is governed by the Operator Precedence and Associativity rules.

The operands in an expression must have either GOOD or UNCERTAIN quality for the expression to be fully evaluated. An operand with a BAD quality immediately stops the evaluation of the remaining part of an expression, and causes the current program to terminate. However, data with BAD quality can be assigned to a local variable without terminating the program.

The following table summarizes the rules for determining the quality of an expression:

Rules for Determining the Quality of an Expression

Quality of the operands	Quality assigned to the result
All GOOD	GOOD
One or more UNCERTAIN, no BAD	UNCERTAIN
One or more BAD	BAD

The timestamp of an expression depends on whether the expression is just a single variable with no operators (e.g. "x"), or it contains at least one operator (e.g. "x+y"). If an expression contains an operator, the resulting timestamp is the current execution time. Otherwise, the variable's timestamp is used.

Note In a simple variable assignment, such as "x=y;", the Timestamp of x will be the same as the Timestamp of y. However, if you would rather use the current time instead, modify this assignment to include at least one operator (e.g. "x=+y;").

The following sections describe each type of expression in detail.

Arithmetic

The following arithmetic operations are supported.

Operator	Description	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulo (remainder after division)	x % y

If all operands in an arithmetic expression are integers or booleans, they are converted to 64-bit signed integers before the expression is evaluated, and the result of the expression is also a 64-bit signed integer. Otherwise, the operands are converted to 64-bit floating point (double) values, and the result is a 64-bit floating point value.

A special rule applies to addition with string operands. If all operands are strings, the result of the addition is a string that is a concatenation of the operands.

Example:

```
x = "123" + "456";       // The result is "123456"
```

To force an arithmetic operation instead, use a numeric value for the first operand.

Example:

```
x = 0 + "123" + "456"; // The result is an integer value 479
x = 0.0 + "1.5" + "2";   // The result is a floating-point value 3.5
```

The Linear Conversion Sample Program also includes examples of arithmetic expressions.

Relational

The result of a relational expression is a value of type bool. The following relational operations are supported.

Operator	Description	Example
<	Less than	x < y
>	Greater than	x > y
<=	Less than or equal to	x <= y
>=	Greater than or equal to	x >= y
==	Equal to	X == y
!=	Not equal to	x != y

If both operands in a relational expression are strings, they are compared using lexicographic (dictionary) order. If both operands are integers or booleans, they are converted to 64-bit signed integers (int) before the expression is evaluated. Otherwise, the operands are converted to 64-bit floating point (double) values.

Logical

The operands in a logical expression are always converted to type bool before the expression is evaluated. The result of the expression is also of type bool. The following logical operations are supported.

Operator	Description	Example
&&	AND	x && y
||	OR	x || y
!	NOT	!x

Bitwise

The operands in a bitwise expression are converted to 64-bit signed integers (int) before the expression is evaluated. The result of the expression is also of type int. The following bitwise operations are supported:

Operator	Description	Example
&	AND	x & 0x100
|	OR	x | 0x1
~	NOT	~x
^	XOR	x ^ 0x01

Operator Precedence and Associativity

C-logic operators follow a strict precedence, which defines the evaluation order of expressions. Operators associate with either the expression to their left or the expression to their right, a property called "associativity". Operators with equal precedence are evaluated left to right in an expression, unless explicitly forced by parentheses.

The following table shows the precedence and associativity of C-logic operators, from highest to lowest precedence.

Precedence	Operator	Name or Meaning	Associativity
1	.	Member selection (for properties)	Left to right
1	[ ]	Array subscript	Left to right
2	~	One's complement	Right to left
2	!	Logical not	Right to left
2	-	Unary minus	Right to left
2	+	Unary plus	Right to left
3	*	Multiplication	Left to right
3	/	Division	Left to right
3	%	Modulo	Left to right
4	+	Addition	Left to right
4	-	Subtraction	Left to right
5	<	Less than	Left to right
5	>	Greater than	Left to right
5	<=	Less than or equal to	Left to right
5	>=	Greater than or equal to	Left to right
6	==	Equality	Left to right
6	!=	Inequality	Left to right
7	&	Bitwise AND	Left to right
8	^	Bitwise exclusive OR	Left to right
9	|	Bitwise inclusive OR	Left to right
10	&&	Logical AND	Left to right
11	||	Logical OR	Left to right
12	=	Assignment	Right to left

Statements

Programs consist of three types of executable statements:

- Conditional Branch (If-Else) Statements
- Assignment (=) Statements
- Return Statements

Conditional Branch (If-Else) Statements

The if-else statement controls conditional branching. The syntax is shown below. If the value of *expression* is true, *statement1* is executed. If the value of expression is false and the optional *else* is present, *statement2* is executed.

Syntax:

```
if (expression)
{
    statement1;
}
[else
{
    statement2;
}]
```

Where:

expression	Any supported expression
statement1	Any executable statement or series of executable statements
statement2	Any executable statement or series of executable statements

Note The program block delimiting characters ('{' and '}') in the "If" and the "else" sections can be omitted if a block consists of a single executable statement.

Example:

```
if(x > y)
{
        z = x;
}
else
        z = y;
```

The Linear Conversion Sample Program also includes examples of conditional branch statements.

Assignment (=) Statements

An assignment statement assigns the result of an expression to a variable.

Syntax:

VarName **=** *expression***;**

Where:

VarName	Variable name
expression	Any supported expression

Example:

```
x = y;
circumference = 2*pi*radius;
```

An assignment to a variable modifies its value, and all of its properties. However, an assignment to an array element does not change the quality, timestamp, or error code for the whole array. An expression assigned to an array element must not have BAD quality, or an exception will be generated and the program will terminate.

An assignment to a property of a local variable modifies only the selected property. If the assigned expression has BAD quality, an exception will be generated and the program will terminate. Assignments to individual properties for ITEM and public variables are not allowed.

Note An assignment to a local variable is not the same as an assignment to its Value property; the variable assignment modifies the Value, and all other properties while an assignment to the Value property modifies only that property.

If *VarName* is a typed local variable, the result of the expression is converted to the variable type before the assignment. If *VarName* is a typed array, the element counts of both arrays must match, however, the lower bound value of the *VarName* is preserved. The assignment persists between program executions, but does not persist between Save & Update Server operations.

If *VarName* is an untyped local variable (VAR), the result is not converted, but instead the variable assumes the expression's data type. The assignment persists between program executions, but does not persist between Save & Update Server operations.

If *VarName* is of type ITEM or public, the assignment will update the server's cache for the associated data item. If *VarName* is a typed array, the element counts of both arrays must match, however, the lower bound value of the *VarName* is preserved. This type of assignment persists between program executions and Save & Update Server operations.

Caution! If *VarName* is of type ITEM and is associated with a bit or register in an external device, the assignment will not initiate a write to that device. Only the cached value in the server will be updated.

Furthermore, it is likely that the assigned data will be overwritten when the server obtains new data from the external device. Therefore, you should assign values to data items only if they are simulated with the Echo simulation selection.

If you need to write to an external device, setup a crosslink that is triggered from your program. Refer to the OPC Crosslink Help for more information.

Return Statements

The return statement ends the program execution and optionally returns a result of an expression.

Syntax:

return [expression]**;**

Where:

expression	Any supported expression

Example:

```
return varResult;   // End the program and return varResult
return;             // End the program without changing the output
```

The result of the returned expression is converted to the data type of the data item associated with the program. Its value is then assigned to that data item. Typically, a program will have one return statement, but may have more than one, or none.

The Maintenance Time Tracking Sample Program is an example of a program that uses multiple return statements.

Math Functions

The supported math functions are:

Abs	Calculates the absolute value of a number
Acos	Calculates the inverse cosine (arc cosine) of a number
Asin	Calculates the inverse sine (arc sine) of a number
Atan	Calculates the inverse tangent (arc tangent) of a number
Ceil	Calculates the smallest integer greater than or equal to a number
Cos	Calculates the cosine of a number
Exp	Calculates the exponential (natural antilogarithm) of a number
Floor	Calculates the greatest integer less than or equal to a number
IsFiniteNumber	Determines whether the given floating-point value is finite
IsValidNumber	Checks a given floating-point value for not a number (NAN)
Ln	Calculates the natural (base *e*) logarithm of a number
Log	Calculates the common (base 10) logarithm of a number
Max	Determines the greater of two numbers
Min	Determines the lesser of two numbers
Pow	Raises one number to the power of another number
Rand	Generates a pseudorandom number
Round	Rounds a value to a specified number of decimal places
Sin	Calculates the sine of a number
Sqrt	Calculates the square root of a number
Tan	Calculates the tangent of a number

Abs

Calculates the absolute value of the argument.

Syntax

int **Abs(**int *Expression***)**

double **Abs(**double *Expression***)**

Where:

Expression	The value for which you want the absolute value.

Return value

Returns the absolute value of *Expression*.

Remarks

For integer arguments, the Abs function returns an integer absolute value. For all other types, the argument is converted to double and the result of type double is returned.

Examples

```
x = Abs(z);
y = Abs(a-b);
```

Acos

Calculates the arccosine (inverse cosine) of the argument.

Syntax

double **Acos(**double *Expression***)**

Where:

Expression	The cosine of the desired angle, limited to the range of 1 to -1.

Return value

Returns the arccosine of *Expression* in the range 0 to π radians.

Remarks

If the value of *Expression* is less than –1 or greater than 1, acos returns an indefinite.

Examples

```
x = Acos(0.3);
y = Acos(z);
```

Asin

Calculates the arcsine (inverse sine) of the argument.

Syntax

double **Asin(**double *Expression***)**

Where:

Expression	The sine of the desired angle, limited to the range of 1 to -1.

Return value

Returns the arcsine of *Expression* in the range $-\pi/2$ to $\pi/2$ radians.

Remarks

If the value of *Expression* is less than –1 or greater than 1, asin returns an indefinite.

Examples

```
x = Asin(0.3);
y = Asin(z);
```

Atan

Calculates the arctangent (inverse tangent) of the argument.

Syntax

double **Atan(**double *Expression***)**

Where:

Expression	The tangent of the desired angle.

Return value

Returns the arctangent of *Expression* in the range $-\pi/2$ to $\pi/2$ radians.

Examples

```
x = Atan(0.3);
y = Atan(z);
```

Ceil

Calculates the smallest integer greater than or equal to the argument.

Syntax

double **Ceil(**double *Expression***)**

Where:

Expression	The value for which you want the ceiling value.

Return value

Returns a double value representing the smallest integer greater than or equal to *Expression*.

Remarks

Note that the return value is of type double, not int.

Examples

```
x = Ceil(2.7);                // Returns 3
y = Ceil(-2.7);               // Returns -2
```

Cos

Calculates the cosine of a specified angle.

Syntax

double **Cos(**double *Angle***)**

Where:

Angle	Desired angle in radians.

Return value

Returns the cosine of *Angle*.

Remarks

If *Angle* is greater than or equal to 2^{63}, or less than or equal to -2^{63}, a loss of significance in the result occurs.

Examples

```
x = Cos(pi/2);
y = Cos(varAngle + varPhase);
```

Exp

Calculates the exponential (natural antilogarithm) of the argument.

Syntax

double **Exp(**double *Expression***)**

Where:

Expression	Desired exponent of *e*.

Return value

Returns the exponential (natural antilogarithm) of the argument. That is, the result is *e* to the power *Expression*, where *e* is the base of the natural logarithm.

Remarks

The value of *Expression* must be in the range of -7.083964e+002 to 7.097827e+002.

Examples

```
x = Exp(3);
y = Exp(z);
```

Floor

Calculates the greatest integer less than or equal to the argument.

Syntax

double **Floor(**double *Expression***)**

Where:

Expression	The value for which you want the floor value.

Return value

Returns a double value representing the greatest integer less than or equal to *Expression*.

Remarks

Note that the return value is of type double, not int.

Examples

```
x = Floor(2.7);                // Returns 2
y = Floor(-2.7);               // Returns -3
```

IsFiniteNumber

Determines whether the given floating-point value is finite.

Syntax

bool **IsFiniteNumber(**double *Expression***)**

Where:

Expression	The floating-point value to be evaluated.

Return value

Returns true if *Expression* is not infinite; that is, if –INF < *Expression* < +INF. It returns false if *Expression* is infinite or NAN.

Remarks

NAN (Not a Number) is a floating-point value representing an undefined or unrepresentable value.

Examples

```
x = IsFiniteNumber(2.5);            // Returns true
```

IsValidNumber

Checks a given floating-point value for not a number (NAN).

Syntax

bool **IsValidNumber(**double *Expression***)**

Where:

Expression	The floating-point value to be evaluated.

Return value

Returns true if *Expression* is not a NAN; otherwise it returns false.

Remarks

NAN (Not a Number) is a floating-point value representing an undefined or unrepresentable value.

Examples

```
x = IsValidNumber(2.5);        // Returns true
```

Ln

Calculates the natural (base *e*) logarithm of the argument.

Syntax

double **Ln(**double *Expression***)**

Where:

Expression	The number for which you want to determine the logarithm.

Return value

Returns the base *e* logarithm of *Expression*.

Remarks

The value of *Expression* must be greater than zero.

Examples

```
x = Ln(27.5);
y = Ln(z);
```

Log

Calculates the common (base 10) logarithm of the argument.

Syntax

double **Log(**double *Expression***)**

Where:

Expression	The number for which you want to determine the logarithm.

Return value

Returns the base-10 logarithm of *Expression*.

Remarks

The value of *Expression* must be greater than zero.

Examples

```
x = Log(y);
pH = -1*Log(varConcH);
```

Max

Compares two numbers and returns the greater of the two.

Syntax

int **Max(**int *Expression1,* int *Expression2***)**

double **Max(**double *Expression1,* double *Expression2***)**

Where:

Expression1, Expression2	The values to compare.

Return value

Returns the value of the larger of the two arguments.

Remarks

For integer arguments, the Max function returns an integer value. For all other types, the argument is converted to double and a result of type double is returned.

Examples

```
x = Max(a,b);
y = Max(varSetPoint,20);
```

Min

Compares two numbers and returns the lesser of the two.

Syntax

int **Min(**int *Expression1,* int *Expression2***)**

double **Min(**double *Expression1,* double *Expression2***)**

Where:

Expression1, Expression2	The values to compare.

Return value

Returns the value of the smaller of the two arguments.

Remarks

For integer arguments, the Min function returns an integer value. For all other types, the argument is converted to double and a result of type double is returned.

Examples

```
x = Min(a,b);
y = Min(Evil1,Evil2);   // Returns the lesser of two evils
```

Pow

Calculates the value of a base number raised to the power of an exponent.

Syntax

double **Pow(**double *Base,* double *Exponent***)**

Where:

Base	The value you want to raise to a power.
Exponent	An expression whose value is the power to which *Base* will be raised.

Return value

Returns the value of *Base* raised to the power of *Exponent*.

Remarks

Pow does not recognize integral floating-point values greater than 2^{64}, such as 1.0E100.

Examples

```
x = Pow(5,3);                   // Returns 125.0
y = Pow(3,5);                   // Returns 243.0
```

Rand

Generates a pseudorandom number.

Syntax

int **Rand()**

Return value

Returns pseudorandom integer in the range 0 to 0x0FFFFFFFF.

Remarks

The Rand function uses the operating system to generate cryptographically secure random numbers.

Examples

```
x = Rand();             // Returns a pseudorandom value
```

Round

Rounds the value to the specified number of decimal places.

Syntax

double **Round(**double *Number***)**

double **Round(**double *Number,* 0**)**

string **Round(**double *Number,* int *Places***)**

Where:

Number	The value to be rounded.
Places	The number of decimal places in the rounded value. Must be non-negative. If not specified, it defaults to 0.

Return value

Returns the value of *Number* rounded to the specified number of decimal places.

Remarks

If *Places* is zero, it will be possible to obtain a floating point value that exactly represents the rounded number, so the return value is of type double.

If *Places* is greater than zero, it may not be possible to represent the exact value in floating point form. Therefore, the return value is of type string.

Examples

```
x = Round(pi,2);              // Returns "3.14"
y = Round(varTemp,1);
```

Sin

Calculates the sine of a specified angle.

Syntax

double **Sin(**double *Angle***)**

Where:

Angle	Desired angle in radians.

Return value

Returns the sine of *Angle*.

Remarks

If *Angle* is greater than or equal to 2^{63}, or less than or equal to -2^{63}, a loss of significance in the result occurs.

Examples

```
x = Sin(2*pi);
y = Sin(varAngle);
```

The Two Sines Sample Program also includes examples of this function.

Sqrt

Calculates the square root of the argument.

Syntax

double **Sqrt(**double *Expression***)**

Where:

Expression	The value for which you want to determine the square root.

Return value

Returns the square root of *Expression*.

Remarks

The value of *Expression* must not be negative.

Examples

```
x = Sqrt(25.);
y = Sqrt(varMeanSquare);
```

The Square Root Conversion Sample Program also includes examples of this function.

Tan

Calculates the tangent of a specified angle.

Syntax

double **Tan(**double *Angle***)**

Where:

Angle	Desired angle in radians.

Return value

Returns the tangent of *Angle*.

Remarks

If *Angle* is greater than or equal to 2^{63}, or less than or equal to -2^{63}, a loss of significance in the result occurs.

Examples

```
x = Tan(pi/4);
y = Tan(varAngle);
```

String Functions

The supported string functions are:

Compare	Compares two strings
Concat	Concatenates two strings
Contains	Determines if a specified string occurs within another string
EndOf	Returns a string that is the end of a specified string
EndsWith	Determines if the end of a string matches another specified string
Format	Returns formatted data as a string
IndexOf	Locates the first occurrence of a specified string within another string
Insert	Inserts a string into another string
Length	Determines the length of a string
Like	Compares strings using wildcards
PadEnd	Left-aligns a string by padding it on the right with a specified character
PadStart	Right-aligns a string by padding it on the left with a specified character
Remove	Removes a specified number of characters at a specified place in a string
Replace	In a given string, replaces a specified string with another specified string
StartOf	Returns a string that is the beginning of a specified string
StartsWith	Determines if the beginning of a string matches another string
Substring	Returns a string that is part of a specified string
ToLower	Converts a string to lowercase
ToNumber	Converts a string to a numeric value
ToString	Converts an integer number to a string
ToUpper	Convert a string to uppercase
Trim	Removes a specified set of characters from both ends of a string
TrimEnd	Removes a specified set of characters from the end of a string
TrimStart	Removes a specified set of characters from the beginning of a string

Compare

Compares two specified String objects, ignoring or honoring their case.

Syntax

int **Compare(**string *StringA,* string *StringB***)**

int **Compare(**string *StringA,* string *StringB,* bool *IgnoreCase***)**

Where:

StringA	First string to compare.
StringB	Second string to compare.
IgnoreCase	A boolean expression indicating whether the comparison is case-sensitive. If true, the comparison is case-insensitive. If not specified, it defaults to false.

Return value

Returns a 64-bit signed integer indicating the lexical (dictionary order) relationship between the two strings.

Value	**Condition**
Less than zero	*StringA* is less than *StringB*.
Zero	*StringA* equals *StringB*.
Greater than zero	*StringA* is greater than *StringB*.

Examples

```
// Compare strings using case-insensitive compare
if(Compare(x, "Name", true) == 0)
{
      // Strings match
      ...
}
```

Concat

Concatenates two strings.

Syntax

string **Concat(**string *String1,* string *String2***)**

Where:

String1, String2	Strings to be concatenated.

Return value

Returns the concatenation of *String1* and *String2*.

Remarks

Two variables of type string can also be concatenated by using a plus sign ('+') in an expression. For example, Concat("abc", "123") is equivalent to an expression: "abc" + "123".

Examples

```
x = Concat("tech ","support");      // Returns "tech support"
```

Contains

Returns a boolean indicating whether the specified *SeekString* occurs within the *String*.

Syntax

bool **Contains(**string *String,* string *SeekString***)**

bool **Contains(**string *String,* string *SeekString,* bool *IgnoreCase***)**

Where:

String	First string to compare.
SeekString	Second string to compare.
IgnoreCase	A boolean expression indicating whether the search is case-sensitive. If true, the search is case-insensitive. If not specified, it defaults to false.

Return value

Returns a boolean indicating whether the specified *SeekString* occurs within the *String*.

Remarks

Returns true if the *SeekString* occurs within the *String*, or if *SeekString* is the empty string (""); otherwise, returns false.

Examples

```
Contains("text", "ex")              // Returns true
Contains("text", "EX")              // Returns false
Contains("text", "EX", true)        // Returns true
Contains("text", "")                // Returns true
```

EndOf

Returns a string that is the end of a specified string.

Syntax

string **EndOf(**string *String,* int *Length***)**

Where:

String	Provided string.
Length	Specifies the number of characters in the resulting string.

Return value

Returns a string equivalent to the substring of length *Length* that ends at the end of *String*.

Examples

```
x = EndOf("Password: XB7A96",6);    // Returns "XB7A96"
```

EndsWith

Determines whether the end of a string matches another specified string.

Syntax

bool **EndsWith(**string *String,* string *EndString***)**

bool **EndsWith(**string *String,* string *EndString,* bool *IgnoreCase***)**

Where:

String	The string to be checked
EndString	A string to compare to.
IgnoreCase	A boolean expression indicating whether the search is case-sensitive. If true, the search is case-insensitive. If not specified, it defaults to false.

Return value

Returns true if *EndString* matches the end of *String*; otherwise returns false.

Remarks

Compares *EndString* to the substring at the end of *String* that is the same length as *EndString*, and indicates whether they are equal. To be equal, *EndString* must be an empty string, or match the end of *String*.

Examples

```
x = EndsWith("MBX Bridge", "Driver");          // Return false
y = EndsWith("MBX Driver", "Driver");          // Returns true
y = EndsWith("MBX Driver", "driver");          // Returns false
y = EndsWith("MBX Driver", "driver", true);    // Returns true
```

Format

Returns formatted data as a string.

Syntax

string **Format(**string *FormatString* [,*Argument*] **...)**

Where:

FormatString	Format-control string.
Argument ...	Optional arguments of type string, sbyte, int16, int32, int64, float, double, byte, uint16, uint32, or uint64.

Return value

Returns a copy of *FormatString* in which the format items have been replaced by the string equivalent of the corresponding *Argument*.

Remarks

Each *Argument* (if any) is converted and output according to the corresponding format specification in *FormatString*. A null character is appended after the last character written. Refer to Appendix C: Format Specification Fields for details on how the format specifications are interpreted.

Examples

```
varString = Format("ItemX = %d\n", x);
```

The Time in Your Time Zone #1 Sample Program also includes examples of this function.

IndexOf

Reports the index of the first occurrence of the specified string within another string. The search starts at a specified character position.

Syntax

int **IndexOf(**string *String,* string *FindString***)**

int **IndexOf(**string *String,* string *FindString,* int *StartIndex***)**

int **IndexOf(**string *String,* string *FindString,* int *StartIndex,* bool *IgnoreCase***)**

Where:

String	The string in which you will search for *FindString*.
FindString	The string you are searching for within *String*.
StartIndex	Indicates where to start the search. This index is zero-based. If not specified, it defaults to 0.
IgnoreCase	A boolean expression indicating whether the search is case-sensitive. If true, the search is case-insensitive. If not specified, it defaults to false.

Return value

Returns the zero-based index of the first occurrence of a specified string within another string, starting from a specified position. If the search fails to locate a match, the function returns a value of -1.

Examples

```
x = IndexOf("text", "ex");          // Returns 1
x = IndexOf("text", "EX");          // Returns -1
x = IndexOf("text", "EX", true);    // Returns 1
```

The ABC to abc Sample Program also includes an example of this function.

Insert

Inserts a string into another string at a specified index position.

Syntax

string **Insert(**string *String,* int *Index,* int *InsertString***)**

Where:

String	The string in which *InsertString* will be inserted.
Index	The position in *String* at which *InsertString* will be inserted.
InsertString	The string to be inserted.

Return value

Returns a new string equivalent to *String* but with *InsertString* inserted at position *Index*.

Remarks

If *Index* is equal to the length of *String*, *InsertString* is appended to the end of *String*.

Examples

```
x = Insert("abc", 2, "XYZ");      // Returns "abXYZc"
y = Insert("abc", 3, "XYZ");      // Returns "abcXYZ"
```

Length

Returns the length of the specified string.

Syntax

int **Length(**string *String***)**

Where:

String	Provided string.

Return value

Returns the number of characters in *String*, not including a terminating null character.

Examples

```
x = Length("Montana");        // Returns 7
y = Length(varFirstName);
```

Like

String compare with wildcards, ignoring or honoring case.

Syntax

bool **Like(**string *String,* string *Pattern***)**

bool **Like(**string *String,* string *Pattern,* bool *IgnoreCase***)**

Where:

String	The string to search in.
Pattern	The pattern to search for.
IgnoreCase	A boolean expression indicating whether the search is case-sensitive. If true, the search is case-insensitive. If not specified, it defaults to false.

Return value

Returns true if a match is found; otherwise returns false.

Remarks

Searches for *Pattern* inside *String* and returns a boolean that indicates whether or not *Pattern* is contained in *String*.

You can use the following symbols in *Pattern* to provide wildcard matches:

- * Accepts zero or more characters.
- ? Accepts any single character.
- # Accepts a single digit (0-9).
- [*CharList*] Accepts a single character if it is part of *CharList*.
- [!*CharList*] Accepts a single character if it is not part of *CharList*.

CharList can specify a range of characters by separating the lower and upper bounds of the range with a hyphen (-). For example, [A-Z] results in a match if the corresponding character position in *String* contains any uppercase letter. You can specify multiple ranges in a single *CharList* (e.g. [A-Z0-9]).

Additional rules for *CharList* are:

- An exclamation point (!) at the beginning of *CharList* results in a match if any character except the characters in *CharList* is found in *String*. When used outside the brackets, the exclamation point matches itself.
- A hyphen (-) at the beginning (after an exclamation point, if one is used) or at the end of *CharList* matches itself. In any other location within *CharList*, the hyphen identifies a range of characters.
- When you use a range of characters, you must specify it in ascending sort order, that is, from lowest to highest. [A-Z] is a valid *CharList*, but [Z-A] is not.
- The character sequence [] is considered a zero-length string ("").

Note To match the special characters left bracket ([), question mark (?), number sign (#), or asterisk (*), enclose them in brackets. You cannot use a right bracket (])within a *CharList* to match itself, but you can use it outside a *CharList* as an individual character.

Note The Like function performs a character-by-character comparison of the *String* and the *Pattern* strings. If these strings have different length, then the function will return false, unless the * (asterisk) is used.

Examples

```
Like("text", "text")          // Returns true
Like("text", "Text")          // Returns false
Like("text", "Text", true)    // Returns true
Like("text", "te*")           // Returns true
Like("text", "te??")          // Returns true
Like("text", "te?")           // Returns false
Like("text", "t?xt")          // Returns true
Like("text", "t[def]xt")      // Returns true
Like("text", "te[abc]t")      // Returns false
Like("text", "[a-z]ext")      // Returns true
Like("text", "[a-z]e*")       // Returns true
Like("text", "[a-z]e")        // Returns false
Like("text", "t[!ab]xt")      // Returns true
Like("text2", "text#")        // Returns true
```

PadEnd

Left-aligns the characters in a string, padding on the right with a specified Unicode character for a specified total length.

Syntax

string **PadEnd(**string *String,* int *TotalLength***)**

string **PadEnd(**string *String,* int *TotalLength,* uint16 *PaddingChar***)**

Where:

String	String to be padded to the given length.
TotalLength	Length of *String* after padding.
PaddingChar	Unicode character to be used to pad *String* to the desired length. If not specified, defaults to space.

Return value

A new string that is equivalent to *String*, but left-aligned and padded on the right with as many *PaddingChar* characters as needed to create a length of *TotalLength*. Or, if *TotalLength* is less than the length of *String*, a new string that is identical to *String*.

Remarks

If *PaddingChar* is not specified, it defaults to a space character.

Examples

```
x = PadEnd("Cyberlogic", 16, " ");      // Returns "Cyberlogic      "
x = PadEnd("Cyberlogic", 16);           // Returns "Cyberlogic      "
y = PadEnd(PartName, 9, "_");
```

PadStart

Right-aligns the characters in a string, padding on the left with a specified Unicode character for a specified total length.

Syntax

string **PadStart(**string *String,* int *TotalLength***)**

string **PadStart(**string *String,* int *TotalLength,* uint16 *PaddingChar***)**

Where:

String	String to be padded to the given length.
TotalLength	Length of *String* after padding.
PaddingChar	Unicode character to be used to pad *String* to the desired length. If not specified, defaults to space.

Return value

A new string that is equivalent to *String*, but right-aligned and padded on the left with as many *PaddingChar* characters as needed to create a length of *TotalLength*. Or, if *TotalLength* is less than the length of *String*, a new string that is identical to *String*.

Remarks

If *PaddingChar* is not specified, it defaults to a space character.

Examples

```
x = PadStart("Cyberlogic", 16, ' ');    // Returns "      Cyberlogic"
x = PadStart("Cyberlogic", 16);         // Returns "      Cyberlogic"
y = PadStart(StationName, 4, '_');
```

Remove

Deletes a specified number of characters from a string beginning at a specified index position.

Syntax

string **Remove(**string *String,* int *Index***)**

string **Remove(**string *String,* int *Index,* int *CharCount***)**

Where:

String	The string in which characters will be deleted.
Index	The position in *String* at which to begin deleting characters.
CharCount	The number of characters to delete.

Return value

A new string that is equivalent to *String* less specified number of characters.

Remarks

If *CharCount* is not specified, this function deletes all the characters from *String* beginning at a specified *Index* position and continuing through the last position.

Examples

```
x = Remove("123abc456", 3, 3);    // Returns "123456"
y = Remove("abc123", 3);          // Returns "abc"
```

Replace

Replaces all occurrences of a specified string in the *Source* string, with another specified string.

Syntax

string **Replace(**string *Source*, string *OldString*, string *NewString***)**

string **Replace(**string *Source*, string *OldString*, string *NewString,* bool *IgnoreCase***)**

Where:

Source	Specifies the string to be modified.
OldString	Specifies the portion of *Source* to be replaced.
NewString	A string to replace all occurrences of *OldString*.
IgnoreCase	A boolean expression indicating whether the search is case-sensitive. If true, the search is case-insensitive. If not specified, it defaults to false.

Return value

Returns a string equivalent to the *Source* string, but with all instances of *OldString* replaced with *NewString*.

Remarks

If *NewString* is an empty string (""), all occurrences of *OldString* are removed.

Examples

```
x = Replace("Switch is open", "open", "closed");
      // The above returns "Switch is closed"
```

StartOf

Returns a string that is the beginning of a specified string

Syntax

string **StartOf(**string *String, int Length***)**

Where:

String	Provided string.
Length	Specifies the number of characters in the resulting string.

Return value

Returns a string equivalent to the substring of length *Length* that begins at start of *String*.

Examples

```
x = StartOf("MBX OPC Server",3);    // Returns "MBX"
```

StartsWith

Determines whether the beginning of a string matches another specified string.

Syntax

bool **StartsWith(**string *String,* string *StartString***)**

bool **StartsWith(**string *String,* string *StartString,* bool *IgnoreCase***)**

Where:

String	The string to be checked.
StartString	A string to compare to.
IgnoreCase	A boolean expression indicating whether the search is case-sensitive. If true, the search is case-insensitive. If not specified, it defaults to false.

Return value

Returns true if *StartString* matches the beginning of *String*; otherwise returns false.

Remarks

Compares *StartString* to the substring at the beginning of *String* that is the same length as *StartString*, and indicates whether they are equal. To be equal, *StartString* must be an empty string, or match the beginning of *String*.

Examples

```
x = StartsWith("DHX Driver", "MBX");           // Returns false
y = StartsWith("MBX Driver", "MBX");           // Returns true
y = StartsWith("MBX Driver", "mbx");           // Returns false
y = StartsWith("MBX Driver", "mbx", true);     // Returns true
```

Substring

Returns a string that is part of a specified string.

Syntax

string **Substring(**string *String,* int *StartIndex***)**

string **Substring(**string *String,* int *StartIndex,* int *Length***)**

Where:

String	Provided string.
StartIndex	Position within *String* of the first character of the desired substring. This index is zero-based. Valid values are between 0 and the length of *String*.
Length	Number of characters in the desired substring.

Return value

Returns a string equivalent to the substring of length *Length* that begins at *StartIndex* in *String*. If *Length* is not specified, the substring will continue through the end of *String*.

Returns an empty string ("") if *Length* is zero, or if *Length* is not specified and *StartIndex* is equal to the length of *String*.

Returns an error if *StartIndex* plus *Length* is greater than the length of *String*.

Examples

```
x = substring("Cyberlogic",5,5);     // Returns "logic"
x = substring("Cyberlogic",5);       // Returns "logic"
```

ToLower

Convert a string to lowercase, using the default system locale.

Syntax

string **ToLower(**string *String***)**

Where:

String	The string to be converted.

Return value

Returns converted *String* in lowercase.

Remarks

The conversion depends on the default system locale, therefore different results may be produced depending on the system language setting.

Examples

```
x = ToLower("DHX OPC Server");      // Returns "dhx opc server"
```

ToNumber

Converts a string to a numeric value according to a specified base.

Syntax

int **ToNumber(**string *String***)**

int **ToNumber(**string *String,* int *Base***)**

int **ToNumber(**string *String,* int *Base,* bool *NumberOnly***)**

Where:

String	The string to be converted to a number.
Base	The base to be used to interpret the string's numeric value. Acceptable values are 0, and 2 through 36. If not specified, it defaults to 0. If *Base* is zero, the *String* must be in the following format, and is interpreted accordingly: [whitespace] [{+ | –}] [0 { x | X | t | T| b | B}] [digits] If { x | X | t | T| b | B} is not specified, the base is decimal.
NumberOnly	A flag that specifies if the *String* must end with characters that are part of the number. If *NumberOnly* is false, the string can end with characters that are not part of the number. If not specified, *NumberOnly* defaults to true.

Return value

Returns the value represented in *String* according to a specified base.

Caution!

If *Base* is 0 or is not specified, then you must use the b|B|t|T|x|X notation to indicate binary, octal or hexadecimal numbers. The specified value is always taken as a 64-bit signed integer, with the most significant bit extended to fill any unspecified bit positions. Therefore, if the value you specify has 1 as the most significant bit, it will be interpreted as a negative number. To force such a value to be interpreted as positive, you must add a leading 0.

For example, "0xFF" will be taken as -1. Its MSB is 1, and that will be extended to produce the 64-bit signed integer FFFFFFFFFFFFFFFF. If you intend it to be taken as 255, you must enter the value as "0x0FF". Doing so forces the MSB to be 0, and the 64-bit representation will then be 00000000000000FF.

This is a concern for binary numbers with a most significant bit of 1, octal numbers with a most significant digit in the range 4-7, and hexadecimal numbers with a most significant digit in the range of 8-F.

This is not an issue for decimal numbers, because they are always assumed to be positive unless the "–" sign is present.

Examples

```
x = ToNumber("FF",16);              // Returns 255
x = ToNumber("0xFF");               // Returns -1
x = ToNumber("0x0FF");              // Returns 255
x = ToNumber("0x0FFzz",0,false);    // Returns 255
y = ToNumber("0b101");              // Returns -3
y = ToNumber("0b0101");             // Returns 5
y = ToNumber("0t30");               // Returns 24
y = ToNumber("0t7");                // Returns -1
y = ToNumber("0t07");               // Returns 7
y = ToNumber(LotNumber,10);
```

ToString

Converts an integer number to a string according to a specified base.

Syntax

string **ToString(**int *Number,* int *Base***)**

Where:

Number	The number to be converted to a string.
Base	The base to be used for the string representation of the *Number*. The value must be in the range of 2-36.

Return value

Returns a string that represents a converted integer *Number* in the specified base.

Examples

```
x = ToString(230,16);         // Returns "E6"
y = ToString(z,10);
```

ToUpper

Convert a string to uppercase, using the default system locale.

Syntax

string **ToUpper(***String***)**

Where:

String	The string to be converted.

Return value

Returns converted *String* in uppercase.

Remarks

The conversion depends on the default system locale, therefore different results may be produced depending on the system language setting.

Examples

```
x = ToUpper("DHX OPC Server");      // Returns "DHX OPC SERVER"
```

Trim

Removes all instances of a specified set of characters from the beginning and end of a specified string.

Syntax

string **Trim(**string *String***)**

string **Trim(**string *String,* string *TrimChars***)**

Where:

String	The string to be trimmed.
TrimChars	A list of characters to remove from the beginning and end of *String*.

Return value

Returns the string that remains after all occurrences of the characters in *TrimChars* are removed from the beginning and end of *String*.

Remarks

The *TrimChars* parameter is optional. If not present, whitespace characters are trimmed instead. The whitespace characters are: space, horizontal tab ('\t'), new line ('\n'), carriage return ('\r'), form feed ('\f') and vertical tab ('\v').

Examples

```
x = Trim("  Adams ", " ");     // Returns "Adams"
x = Trim("  Adams ");          // Returns "Adams"
x = Trim("12Adams31", "123"); // Returns "Adams"
```

TrimEnd

Removes all instances of a specified set of characters from the end of a specified string.

Syntax

string **TrimEnd(**string *String***)**

string **TrimEnd(**string *String,* string *TrimChars***)**

Where:

String	The string to be trimmed.
TrimChars	A list of characters to remove from the end of *String*.

Return value

Returns the string that remains after all occurrences of the characters in *TrimChars* are removed from the end of *String*.

Remarks

The *TrimChars* parameter is optional. If not present, whitespace is trimmed instead. The whitespace characters are: space, horizontal tab ('\t'), new line ('\n'), carriage return ('\r'), form feed ('\f') and vertical tab ('\v').

Examples

```
x = TrimEnd("  Adams ", " ");        // Returns "  Adams"
x = TrimEnd("  Adams ");             // Returns "  Adams"
x = TrimEnd("12Adams31", "123");     // Returns "12Adams"
```

TrimStart

Removes all instances of a specified set of characters from the beginning of a specified string.

Syntax

string **TrimStart(**string *String***)**

string **TrimStart(**string *String,* string *TrimChars***)**

Where:

String	The string to be trimmed.
TrimChars	A list of characters to remove from the beginning of *String*.

Return value

Returns the string that remains after all occurrences of the characters in *TrimChars* are removed from the beginning of *String*.

Remarks

The *TrimChars* parameter is optional. If not present, whitespace characters are trimmed instead. The whitespace characters are: space, horizontal tab ('\t'), new line ('\n'), carriage return ('\r'), form feed ('\f') and vertical tab ('\v').

Examples

```
x = TrimStart("  Adams ", " ");      // Returns "Adams "
x = TrimStart("  Adams ");           // Returns "Adams "
x = TrimStart("12Adams31", "123");  // Returns "Adams31"
```

Bitwise Functions

The supported bitwise functions are:

GetBitField	Obtains the value of a specified field of bits in a number
SAR	Shifts the bits of a number right, filling the left with the sign bit
SHL	Shifts the bits of a number left, filling the right with zeros
SHR	Shifts the bits of a number right, filling the left with zeros

GetBitField

Returns the value of a specified field of bits within a number.

Syntax

int **GetBitField(**int *Number,* int *StartBit,* int *BitCount***)**

Where:

Number	Number containing the desired bit field.
StartBit	Zero-based index of the first bit of the desired bit field.
BitCount	Number of bits in the desired bit field.

Return value

Returns the value of a specified field of bits within a number.

Remarks

This function shifts the specified *Number* right by the *StartBit* count, and then masks off the high bits above the LS *BitCount* bits. As a result, the masked off high bits are always cleared, and the returned result is not sign extended.

Examples

```
x = GetBitField(0xE6,2,4);    // Returns 9
y = GetBitField(z,5,2);
```

SAR

Performs an arithmetic shift right of a numeric value.

Syntax

int **SAR(**int *Number,* int *ShiftByCount***)**

Where:

Number	Value to be shifted.
ShiftByCount	Number of places to shift the bits of *Number*.

Return value

Returns an integer number that is the result of the specified arithmetic right shift.

Remarks

The SAR (shift arithmetic right) function shifts the bits in the *Number* argument to the right (toward the less significant bit locations). For each shift count, the most significant bit of the result is filled with the sign of the unshifted *Number*, and the least significant bit is shifted out.

Examples

```
x = SAR(-4, 1);         // Returns -2
y = SAR(-20, 2);        // Returns -5
```

SHL

Performs a logical shift left of a numeric value.

Syntax

int **SHL(**int *Number,* int *ShiftByCount***)**

Where:

Number	Value to be shifted.
ShiftByCount	Number of places to shift the bits of *Number*.

Return value

Returns an integer number that is the result of the specified logical left shift.

Remarks

The SHL (shift logical left) function shifts the bits in the *Number* argument to the left (toward the more significant bit locations). For each shift count, the most significant bit of the result is shifted out, and the least significant bit is cleared.

Examples

```
x = SHL(1, 1);          // Returns 2
y = SHL(5, 2);          // Returns 20
```

SHR

Performs a logical shift right of a numeric value.

Syntax

int **SHR(**int *Number,* int *ShiftByCount***)**

Where:

Number	Value to be shifted.
ShiftByCount	Number of places to shift the bits of *Number*.

Return value

Returns an integer number that is the result of the specified logical right shift.

Remarks

The SHR (shift logical right) function shifts the bits in the *Number* argument to the right (toward the less significant bit locations). For each shift count, the most significant bit of the result is cleared, and the least significant bit is shifted out.

Examples

```
x = SHR(4, 1);          // Returns 2
y = SHR(20, 2);         // Returns 5
```

Date & Time Functions

The supported date and time functions are:

AddSeconds	Adds an interval in seconds to a DATETIME variable
FormatDate	Formats a DATETIME value as a string containing the date
FormatTime	Formats a DATETIME value as a string containing the time
GetTimeZoneOffset	Provides the time zone offset in seconds
TimeNow	Provides the current time

AddSeconds

Adds an interval in seconds to a specified variable of type DATETIME.

Syntax

DATETIME **AddSeconds(**DATETIME *DateTime,* double *Seconds***)**

Where:

DateTime	Date and time value.
Seconds	The number of seconds you want to add to *DateTime*.

Return value

Returns a DATETIME-compatible UTC value that represents the date and time after *Seconds* have been added to *DateTime*.

Remarks

When a DATETIME variable is accessed directly, it represents an OPC timestamp-compatible 64-bit signed integer (int64) value. Therefore, it is a common practice to pass an integer value for the *DateTime* parameter.

Examples

```
// Add 10.5 seconds to the item's timestamp
x.Timestamp = AddSeconds(x.Timestamp, 10.5);
```

The Time in Your Time Zone #1 Sample Program also includes examples of this function.

FormatDate

Formats a value of type DATETIME as a string containing the date in the local locale.

Syntax

string **FormatDate(**DATETIME *DateTime***)**

Where:

DateTime	Date and time value.

Return value

Returns a string representing the date in the local locale.

Remarks

The date is always formatted according to the locale (language) for the system and represents the local date, even if the *DateTime* variable was configured for the UTC time zone.

When a DATETIME variable is accessed directly, it represents an OPC timestamp-compatible 64-bit signed integer (int64) value. Therefore, it is a common practice to pass an integer value for the *DateTime* parameter.

Examples

```
// Format the date of x as a string and put it in varDate
varDate = FormatDate(x.Timestamp);
```

FormatTime

Formats a value of type DATETIME as a string containing the time in the local locale.

Syntax:

string **FormatTime(**DATETIME *DateTime***)**

Where:

DateTime	Date and time value.

Return value

Returns a string representing the time in the local locale.

Remarks

The time is always formatted according to the locale (language) for the system and represents the local time, even if the *DateTime* variable was configured for the UTC time zone.

When a DATETIME variable is accessed directly, it represents an OPC timestamp-compatible 64-bit signed integer (int64) value. Therefore, it is a common practice to pass an integer value for the *DateTime* parameter.

Examples

```
// Format the time of x as a string and put it in varTime
varTime = FormatTime(x.Timestamp);
```

GetTimeZoneOffset

Returns the local time zone offset from UTC (Coordinated Universal Time).

Syntax

double **GetTimeZoneOffset()**

Return value

Returns the local time zone offset in seconds.

Remarks

GetTimeZoneOffset uses the current settings for the time zone and daylight saving time. Therefore, when daylight saving time is in effect, this function will take it into account.

Examples

```
// This program returns a string showing the GMT time,
// using the system locale settings
int dt;                            // Current local time

// Read current time
dt = TimeNow();

// Subtract the time zone offset
dt = AddSeconds(dt, -GetTimeZoneOffset());

// Format the output string using your current locale settings
return FormatDate(dt) + " " + FormatTime(dt);
```

TimeNow

Provides the current time.

Syntax

DATETIME **TimeNow()**

Return value

Returns a DATETIME-compatible UTC value that represents the current time.

Remarks

When a DATETIME variable is accessed directly, it represents an OPC timestamp-compatible 64-bit signed integer (int64) value. Therefore, it is possible to assign the result of this function directly to an int64 variable or to a timestamp property.

Examples

```
y.Timestamp = TimeNow();      // Change timestamp to current time
```

The Time in Your Time Zone #1 Sample Program and Two Sines Sample Program also include examples of this function.

Variable Properties Functions

All C-logic variables include Error and Quality properties. The C-logic language includes Error and OPC Quality functions, which simplify the interpretation of these properties.

The valid error code values are the same as the Microsoft HRESULT error codes. These are 32-bit values with several fields encoded within the value. They can represent either success or failure conditions.

The valid quality values are the same as the OPC Foundation quality codes. Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

The supported Error and OPC Quality functions are:

GetErrorString	Returns the error string for a variable or constant
IsErrorFAILURE	Determines if the error code of the argument is a failure
IsErrorSUCCESS	Determines if the error code of the argument is a success
IsQualityBAD	Determines if the quality is BAD
IsQualityGOOD	Determines if the quality is GOOD
IsQualityUNCERTAIN	Determines if the quality is UNCERTAIN
QualityLimitField	Obtains the quality limit bits for a variable
QualityStatusCode	Obtains the quality without limit bits for a variable

GetErrorString

Returns the error string for the Error code of the specified argument.

Syntax

string **GetErrorString(***Argument***)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns the error string associated with the Error code of *Argument*.

Remarks

The GetErrorString function uses the default system locale.

Examples

```
ErrorStr = GetErrorString(x);
```

IsErrorFAILURE

Determines if the Error code of the specified argument represents a failure.

Syntax

bool **IsErrorFAILURE(***Argument***)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns true if the Error property of *Argument* represents a failure.

Remarks

Negative Error property values indicate failure.

Examples

```
if(IsErrorFAILURE(x))
{
      return;
}
```

IsErrorSUCCESS

Determines if the Error code of the specified argument represents a success.

Syntax

bool **IsErrorSUCCESS(***Argument***)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns true if the Error property of *Argument* represents a success.

Remarks

Non-negative Error property values indicate success.

Examples

```
if(IsErrorSUCCESS(x))
{
      varX = x;
}
```

IsQualityBAD

Determines if the quality of the specified argument is BAD. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

Syntax

bool **IsQualityBAD(***Argument* **)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns true if the Quality property of *Argument* is BAD.

Remarks

The quality of a constant is always GOOD.

Examples

```
if(IsQualityBAD(x))
{
        varResult.Quality = QUALITY_SENSOR_FAILURE;
}
```

The Maintenance Time Tracking Sample Program also includes examples of how to test the Quality property.

IsQualityGOOD

Determines if the quality of the specified argument is GOOD. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

Syntax

bool **IsQualityGOOD(***Argument***)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns true if the Quality property of *Argument* is GOOD.

Remarks

The quality of a constant is always GOOD.

Examples

```
if(IsQualityGOOD(x))
{
        varX = x;
}
```

The Maintenance Time Tracking Sample Program also includes examples of how to test the Quality property.

IsQualityUNCERTAIN

Determines if the quality of the specified argument is UNCERTAIN. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

Syntax

bool **IsQualityUNCERTAIN(***Argument* **)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns true if the Quality property of *Argument* is UNCERTAIN.

Remarks

The quality of a constant is always GOOD.

Examples

```
if(IsQualityUNCERTAIN(x))
{
	x.Quality = QUALITY_BAD;	// Treat UNCERTAIN as BAD
}
```

The Maintenance Time Tracking Sample Program also includes examples of how to test the Quality property.

QualityLimitField

Returns the limit bits of the quality for the selected variable. (Refer to Appendix B: OPC Quality Flags for information about OPC data quality.)

Syntax

int **QualityLimitField(***Argument***)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns the limit bits of the quality for *Argument*.

Remarks

The operation performed by this function is equivalent to the following line of code:

```
Argument.Quality & 0x03;
```

The value returned by this function is typically used in comparisons to the following Predefined Constants:

QUALITY_LIMIT_OK

QUALITY_LIMIT_LOW

QUALITY_LIMIT_HIGH

QUALITY_LIMIT_CONST

Examples

```
if(QualityLimitField(x) == LIMIT_HIGH)
{
      // Treat values at high limit as BAD
      x.Quality = QUALITY_BAD;
}
```

QualityStatusCode

Returns the quality with no limit bits for the specified argument. Refer to Appendix B: OPC Quality Flags for information about OPC data quality.

Syntax

int **QualityStatusCode(***Argument***)**

Where:

Argument	Variable of any type, or constant.

Return value

Returns the quality code with no limit bits for *Argument*.

Remarks

The operation performed by this function is equivalent to the following line of code:

```
Argument.Quality & 0xfc;
```

The value returned by this function is typically used in comparisons to the following predefined constants:

QUALITY_GOOD
QUALITY_BAD
QUALITY_NOT_CONNECTED
QUALITY_SENSOR_FAILURE
QUALITY_COMM_FAILURE
QUALITY_WAITING_FOR_INITIAL_DATA
QUALITY_LAST_USABLE
QUALITY_EGU_EXCEEDED
QUALITY_LOCAL_OVERRIDE
QUALITY_CONFIG_ERROR
QUALITY_DEVICE_FAILURE
QUALITY_LAST_KNOWN
QUALITY_OUT_OF_SERVICE
QUALITY_UNCERTAIN
QUALITY_SENSOR_CAL
QUALITY_SUB_NORMAL

Examples

```
if(QualityStatusCode(x) == QUALITY_LAST_USABLE)
{
      // Treat QUALITY_LAST_USABLE as GOOD
      x.Quality = QUALITY_GOOD;
}
```

Other Functions

Additional functions are:

ArrayBounds	Returns an array with the lower bound values for each array dimension
ArrayDimElements	Returns an array with the number of elements for each array dimension
DebugOutput	Sends a formatted data string to a debug output
GetFixedInterval	Returns the fixed interval used for running the program
IndexSort	Ranks array elements according to either ascending or descending order
IsArray	Determines if the Argument is an array
SetFixedInterval	Sets the fixed interval for running the program to a specified value
Sort	Sorts the input array as specified

ArrayBounds

Returns an array containing the lower-bound values for each dimension of a given array.

Syntax

int[] **ArrayBounds(***ArrayVariable***)**

Where:

ArrayVariable	Array variable.

Return value

If *ArrayVariable* is an array, this function returns a one-dimensional int array containing the lower bound values for each dimension of *ArrayVariable*. Otherwise, a runtime error is generated. The returned array's lower bound is always zero.

Remarks

Currently, C-logic supports only one-dimensional arrays.

Examples

```
double Array[10];
var ArrayBounds;

ArrayBounds = ArrayBounds(Array);
x = ArrayBounds[0];           // x is set to 0
```

ArrayDimElements

Returns an array containing the number of elements for each dimension of a given array.

Syntax

int[] **ArrayDimElements(***ArrayVariable***)**

Where:

ArrayVariable	Array variable.

Return value

If *ArrayVariable* is an array, this function returns a one-dimensional int array containing the number of elements for each dimension of *ArrayVariable*. Otherwise, a runtime error is generated. The returned array's lower bound is always zero.

Remarks

Currently, C-logic supports only one-dimensional arrays.

Examples

```
double Array[10];
var DimElements;

DimElements = ArrayDimElements(Array);
x = DimElements[0];           // x is set to 10
```

DebugOutput

Sends a formatted data string to the selected debug output.

Syntax

string **DebugOutput(**int *OutputIndex*, string *FormatString* [,*Argument*] **...)**

Where:

OutputIndex	Selects a debug output data item located in the *${Item Name}* folder, which is located in the same folder as the program's data item. It must be an integer value in the range of 0-99. The name of the debug output will be "DebugOutputXX", where XX is the two-digit decimal representation of *OutputIndex*.
FormatString	Format-control string.
Argument ...	Optional arguments of type string, sbyte, int16, int32, int64, float, double, byte, uint16, uint32 or uint64.

Return value

Returns a copy of *FormatString* in which each format item has been replaced by the string equivalent of the corresponding *Argument*.

Remarks

This function uses the same format-control string specification as the Format function (Refer to Appendix C: Format Specification Fields for complete information on the format specifications.)

However, in addition to returning a formatted string, it writes the string to the selected debug output. The debug outputs are automatically created in the *${Item Name}* folder, which is located in the same folder as the program's data item. The number of debug outputs depends on the number of *OutputIndex* values used in the program. The name of a debug output is "DebugOutputXX", where XX is a two-digit decimal representation of the corresponding *OutputIndex.* (The *OutputIndex* values 0-9 will have leading zeros appended to them.)

Examples

```
DebugOutput(1, "Old value = %f", x);
```

The Using Debug Outputs Sample Program also includes examples of this function.

GetFixedInterval

Returns the value of the current fixed interval used for running the program.

Syntax

int **GetFixedInterval()**

Return Value

Returns the value of the fixed interval, in milliseconds, that is used to schedule program execution. If the fixed interval is not defined, it returns 0.

Remarks

Each C-logic program can be statically configured to run at a fixed interval. (Refer to the Settings Tab help for more information.) A program can also dynamically set or modify this interval at runtime by calling the SetFixedInterval function. The GetFixedInterval function returns the current setting for this interval.

Examples

```
x = GetFixedInterval();
```

The Maintenance Time Tracking Sample Program also includes examples of this function.

IndexSort

Ranks the elements of an array in ascending or descending order.

Syntax

int[] **IndexSort(***ArrayVariable***)**

int[] **IndexSort(***ArrayVariable,* bool *Descending***)**

int[] **IndexSort(***ArrayVariable,* bool *Descending,* bool *IgnoreCase***)**

Where:

ArrayVariable	Array variable or string.
Descending	A boolean expression indicating the ranking order. If true, the descending order is used. If not specified, it defaults to false.
IgnoreCase	A boolean expression indicating whether case should be taken into account when ranking string values. If true, case is ignored and the ranking is case-insensitive. If not specified, it defaults to false.

Return value

If *ArrayVariable* is a one-dimensional array or a string, the function returns an array of indexes to *ArrayVariable*. Otherwise, it generates a runtime error.

The order of the returned indexes indicates the order of the elements of *ArrayVariable* if they were sorted as specified.

Remarks

The returned array's lower bound is zero. The returned indexes incorporate *ArrayVariable's* lower-bound value. Because C-logic allows strings to be viewed as arrays, it is possible to rank a string. When ranking a string, the *IgnoreCase* parameter is valid. Strings are always considered to have a lower-bound of zero, and so that is used for the returned indexes.

Examples

```
int Array[3]({3,1,2});
var Ranking;

Ranking = IndexSort(Array);         // Ranking = {1, 2, 0}
Ranking = IndexSort(Array, true);   // Ranking = {0, 2, 1}
```

The Rank Machine Performance Sample Program also includes an example of this function.

IsArray

Determines if the argument is an array.

Syntax

int **IsArray(***Argument***)**

Where:

Argument	Variable of any type.

Return value

If *Argument* is an array, this function returns the number of dimensions in that array. Otherwise, it returns 0.

Remarks

Currently, C-logic supports only one-dimensional arrays.

Examples

```
double Array[10]({0});        // Local array
int y;                        // Local variable

x = IsArray(Array);           // Returns 1
x = IsArray(y);               // Returns 0
```

SetFixedInterval

Sets the fixed interval for running the program to a specified value.

Syntax

int **SetFixedInterval(**int *FixedInterval***)**

Where:

FixedInterval	Specifies the fixed interval to be set. The value is in milliseconds and must be in the range 0 - 4000000000. A value of 0 disables the fixed interval execution.

Return Value

Returns the previous value of the fixed interval, in milliseconds, before the new *FixedInterval* value was assigned. If the fixed interval was not defined, it returns 0.

Remarks

Each C-logic program can be statically configured to run at a fixed interval. (Refer to the Settings Tab help for more information.) A program can also dynamically set or modify this interval at runtime by calling the SetFixedInterval function. The program's data item need not be statically configured to run at a fixed interval for the SetFixedInterval function to succeed. Setting the interval to 0 disables the fixed interval execution.

Examples

```
SetFixedInterval(1000); // Set fixed interval execution to 1 sec
SetFixedInterval(0);    // Disable fixed interval execution
```

The Maintenance Time Tracking Sample Program also includes examples of this function.

Sort

Sorts the input array as specified.

Syntax

SortedArray **Sort(***ArrayVariable***)**

SortedArray **Sort(***ArrayVariable,* bool *Descending***)**

SortedArray **Sort(***ArrayVariable,* bool *Descending,* bool *IgnoreCase***)**

Where:

ArrayVariable	Array variable or string.
Descending	A boolean expression indicating the sort order. If true, the descending order is used. If not specified, it defaults to false.
IgnoreCase	A boolean expression indicating whether case should be taken into account when sorting string values. If true, case is ignored and the sort is case-insensitive. If not specified, it defaults to false.

Return value

If *ArrayVariable* is a single-dimensional array, it returns a sorted array. If *ArrayVariable* is a string, it returns a string. Otherwise, a runtime error is generated.

Remarks

The returned array's lower bound matches the lower bound of the original *ArrayVariable*. Because C-logic allows strings to be viewed as arrays, it is possible to sort a string. When sorting a string, the *IgnoreCase* parameter is valid.

Examples

```
int Array[3]({3,1,2});
var Sorted;

Sorted = Sort(Array);          // Sorted = {1, 2, 3}
Sorted = Sort(Array, true);    // Sorted = {3, 2, 1}
```

APPENDIX B: OPC QUALITY FLAGS

The quality flags represent the quality state of the item's data value. It is similar to the Fieldbus Data Quality Specification (section 4.4.1 in the H1 Final Specifications). This makes it easy for both servers and client applications to determine how much functionality they want to implement.

The low eight bits of the Quality flags are defined in the form of three bit fields; Quality, Substatus and Limit status. The Quality bits are arranged as follows:

QQSSSSLL

The high eight bits of the Quality Word are available for vendor-specific use. If these bits are used, the standard OPC Quality bits must still be set as accurately as possible to indicate what assumptions the client can make about the returned data. In addition, it is the responsibility of any client interpreting vendor specific quality information to ensure that the server providing it uses the same rules as the client. The details of such a negotiation are not specified in this standard, although a QueryInterface call to the server for a vendor specific interface such as IMyQualityDefinitions is a possible approach.

The following sections provide details of the OPC standard quality bits.

The Quality Bit Field

QQ	Bit Value	Definition	Description
0	00SSSSLL	Bad	Value is not useful for the reasons indicated by the Substatus.
1	01SSSSLL	Uncertain	The quality of the value is uncertain for the reasons indicated by the Substatus.
2	10SSSSLL	N/A	Not used by OPC
3	11SSSSLL	Good	The quality of the value is good.

A server which supports no quality information must return 3 (Good). It is also acceptable for a server to simply return Bad or Good (0x00 or 0xC0), and to always return 0 for Substatus and Limit.

Clients should check the Quality bit field of all results, even if they do not check the Substatus or Limit fields.

The contents of the Value field must be a well-defined VARIANT even if the Quality is BAD, indicating that it does not contain an accurate value. This simplifies error handling in client applications. For example, clients are always expected to call VariantClear() on the results of a Synchronous Read. Similarly the IAdviseSink must be able to interpret and unpack the Value and Data included in the Stream, even if that data is BAD.

If the server has no known value to return, then it must return a reasonable default value, such as a NUL string or a 0 numeric value.

The Substatus Bit Field

The layout of this field depends on the value of the Quality Field.

Substatus for BAD Quality:

SSSS	Bit Value	Definition	Description
0	000000LL	Non-specific	The value is bad but no specific reason is known.
1	000001LL	Configuration Error	There is some server-specific problem with the configuration. For example, the item in question has been deleted from the configuration.
2	000010LL	Not Connected	The input is required to be logically connected to something, but it is not. This quality may reflect that no value is available at this time, for reasons such as the value not having been provided by the data source.
3	000011LL	Device Failure	A device failure has been detected.
4	000100LL	Sensor Failure	A sensor failure had been detected. The Limits field can provide additional diagnostic information in some situations.
5	000101LL	Last Known Value	Communications have failed, but the last-known value is available. The age of the value may be determined from the TIMESTAMP in the OPCITEMSTATE.
6	000110LL	Comm Failure	Communications have failed. There is no available last-known value.
7	000111LL	Out of Service	The block is off scan or otherwise locked. This quality is also used when the active state of the item, or the group containing the item, is InActive.
8-15		N/A	Not used by OPC.

Servers that do not support Substatus should return 0. Note that an old value may be returned with the Quality set to BAD (0) and the Substatus set to 5. This is for consistency with the Fieldbus Specification. This is the only case in which a client may assume that a BAD value is still usable by the application.

Substatus for UNCERTAIN Quality:

SSSS	Bit Value	Define	Description
0	010000LL	Non-specific	There is no specific reason why the value is uncertain.
1	010001LL	Last Usable Value	Whatever was writing this value has stopped doing so. The returned value should be regarded as stale. The age of the value can be determined from the TIMESTAMP in OPCITEMSTATE. Note that this differs from a BAD value with Substatus 5 (Last Known Value). That status is associated specifically with a detectable communications error on a fetched value. This error is associated with the failure of some external source to put something into the value within an acceptable period of time.
2-3		N/A	Not used by OPC
4	010100LL	Sensor Not Accurate	Either the value is pegged at one of the sensor limits (in which case the limit field should be set to 1 or 2), or the sensor's internal diagnostics have indicated that it is out of calibration (in which case the limit field should be 0).
5	010101LL	Engineering Units Exceeded	The returned value is outside the limits defined for this parameter. Note that in this case (per the Fieldbus Specification) the Limits field indicates which limit has been exceeded but does not necessarily imply that the value cannot move farther out of range.
6	010110LL	Sub-Normal	The value is derived from multiple sources and has less than the required number of Good sources.
7-15		N/A	Not used by OPC.

Servers that do not support Substatus should return 0.

Substatus for GOOD Quality:

SSSS	Bit Value	Define	Description
0	110000LL	Non-specific	The value is good. There are no special conditions.
1-5		N/A	Not used by OPC.
6	110110LL	Local Override	The value has been Overridden. Typically this is means the input has been disconnected and a manually entered value has been forced.
7-15		N/A	Not used by OPC.

Servers that do not support Substatus should return 0.

The Limit Bit Field

The Limit Field is valid regardless of the Quality and Substatus. In some cases, such as Sensor Failure, it can provide useful diagnostic information.

SSSS	Bit Value	Define	Description
0	QQSSSS00	Not Limited	The value is free to move up or down.
1	QQSSSS01	Low Limited	The value has pegged at some low limit.
2	QQSSSS10	High Limited	The value has pegged at some high limit.
3	QQSSSS11	Constant	The value is a constant and cannot move.

Servers that do not support Limit should return 0.

APPENDIX C: FORMAT SPECIFICATION FIELDS

The first parameter in the Format function specifies a format specification. It consists of optional and required fields, and has the following form:

%[flags] [width] [.precision] [{h | w | l | L | ll | I | I32 | I64}]type

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a type character (for example, %s). If a percent sign is followed by a character that has no meaning as a format field, the character is copied as is. For example, to include a percent-sign character in the output string, use %%.

The optional fields, which appear before the type character, control other aspects of the formatting, as follows:

flags

These are optional characters that control justification of output and treatment of signs, blanks, decimal points, and octal and hexadecimal prefixes. Refer to the flag characters table in the Flag Directives section for a list and definitions. More than one flag can appear in a format specification.

width

This is an optional number that specifies the minimum number of characters output. Refer to the Width Specification section for details.

precision

This is an optional number that specifies the maximum number of characters output for all or part of the output field, or the minimum number of digits output for integer values. For details, refer to the table in the Precision Specification section.

h | w | l | L | ll | I | I32 | I64

These are optional prefixes to *type* that specify the size of the argument. Refer to the prefixes table in Size and Distance Specification.

type

Required character that determines whether the associated argument is interpreted as a character, a string, or a number. Refer to the table in Type Field Characters for details.

Flag Directives

The first optional field of the format specification is flags. A flag directive is a character that justifies output and controls the output of signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification.

Flag	Meaning	Default
–	Left align the result within the given field width.	Right align.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative signed values (–).
0	If *width* is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format (i, u, x, X, o, d) and a precision specification is also present (for example, `%04.d`), the 0 is ignored.	No padding.
blank (' ')	Prefix the output value with a blank if the output value is signed and positive. The blank is ignored if both the blank and + flags appear.	No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No prefix appears.
	When used with the e, E, f, a or A format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros.	Decimal point appears only if digits follow it. Trailing zeros are truncated.
	Ignored when used with c, d, i, u, or s.	

Width Specification

The width specification is a nonnegative decimal integer that controls the minimum number of characters output.

If the number of characters in the output value is less than the specified width, blanks are added until the minimum width is reached. If the "–" (left alignment) flag is specified, the blanks are appended to the right of the value; otherwise, they are appended to the left. If width is prefixed with 0, zeros are added until the minimum width is reached.

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if width is not specified, all characters of the value are output, subject to the precision specification.

Precision Specification

The precision specification indicates the number of characters to be output, the number of decimal places, or the number of significant digits. It is a nonnegative decimal integer, preceded by a period (.). Refer to the table for details on how the precision specification is used with each data type.

Unlike the width specification, the precision specification can cause either truncation of the output value or rounding of a floating-point value. If precision is specified as 0 and the value to be converted is 0, the result is no characters output, as shown below:

```
Format( "%.0d", 0 );      // No characters output
```

The type determines the interpretation of precision, and the default when precision is omitted, as shown in the following table.

Type	Meaning	Default
a, A	The precision specifies the number of digits after the point.	Default precision is 6. If precision is 0, no point is output unless the # flag is used.
c, C	The precision has no effect.	Character is output.
d, i, u, o, x, X	The precision specifies the minimum number of digits to be output. If the number of digits in the argument is less than *precision*, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds *precision*.	Default precision is 1.
e, E	The precision specifies the number of digits to be output after the decimal point. The last output digit is rounded.	Default precision is 6; if *precision* is 0 or the period (.) appears without a number following it, no decimal point is output.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if *precision* is 0, or if the period (.) appears without a number following it, no decimal point is output.
g, G	The precision specifies the maximum number of significant digits output.	Six significant digits are output, with any trailing zeros truncated.
s, S	The precision specifies the maximum number of characters to be output. Characters in excess of *precision* are not output.	Characters are output until a null character is encountered.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or NAN, Format gives the following output.

Value	**Output**
+ infinity	1.#INF*random-digits*
– infinity	–1.#INF*random-digits*
Indefinite (same as quiet NAN)	*digit.#*IND*random-digits*
NAN	*digit.#*NAN*random-digits*

Size and Distance Specification

The optional prefixes to type, **h**, **l**, **I**, **I32**, **I64** and **ll** specify the size of the argument (long or short, 32- or 64-bit, single-byte character or wide character, depending upon the type specifier that they modify). These type-specifier prefixes are used with type characters in the Format function to specify interpretation of arguments, as shown in the following table.

To specify	Use prefix	With type specifier
int32	l (lowercase L)	d or i
uint32	l	o, u, x, or X
int64	ll	d, i, o, x, or X
int16	h	d or i
uint16	h	o, u, x, or X
int32	I32	d or i
uint32	I32	o, u, x, or X
int64 (int)	I64	d or i
uint64	I64	o, u, x, or X
int32	I	d or i
uint32	I	o, u, x, or X
double	l or L	f
sbyte	h	c or C
int16	l	c or C
string	l	s or S
int16	w	c
string	w	s

Thus to output single-byte or wide-characters with Format function, use format specifiers as follows.

To output character as	With format specifier
single byte	C, hc, or hC
wide	c, lc, lC, or wc

Type Field Characters

The type character is the only required format field; it appears after any optional format fields. The type character determines whether the associated argument is interpreted as a character, string, or number.

Character	Type	Output format
c	int16	Specifies a wide character.
C	sbyte	Specifies a single-byte character.
d	int32	Signed decimal integer.
i	int32	Signed decimal integer.
o	uint32	Unsigned octal integer.
u	uint32	Unsigned decimal integer.
x	uint32	Unsigned hexadecimal integer, using "abcdef".
X	uint32	Unsigned hexadecimal integer, using "ABCDEF".
e	double	Signed value having the form [–]d.dddd e [*sign*]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is three decimal digits, and *sign* is + or –.
E	double	Identical to the e format except that E rather than e introduces the exponent.
f	double	Signed value having the form [–]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision.
g	double	Signed value output in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than –4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it.
G	double	Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate).

a	double	Signed hexadecimal double precision floating point value having the form [−]0x*h.hhhh* p±ddd, where *h.hhhh* are the hex digits (using lower case letters) of the mantissa, and ddd are three digits for the exponent. The precision specifies the number of digits after the point.
A	double	Signed hexadecimal double precision floating point value having the form [−]0X*h.hhhh* P±ddd, where *h.hhhh* are the hex digits (using upper case letters) of the mantissa, and ddd are three digits for the exponent. The precision specifies the number of digits after the point.
s	string	Specifies a wide-character string. Characters are output up to the first null character or until the *precision* value is reached.
S	string	Specifies a wide-character string. Characters are output up to the first null character or until the *precision* value is reached.

Note All exponential formats use three digits in the exponent.

APPENDIX D: SAMPLE PROGRAMS

All Cyberlogic OPC servers include a configuration file with a set of sample C-logic programs. These will help you to understand how the various functions work. They will also give you ideas about what you can do with C-logic, and they can be modified and used in creating your own programs.

In this appendix, we will discuss some of these programs. For additional sample programs, refer to the *Math & Logic Sample Configuration.mdb* file.

The programs in this appendix are:

ABC to abc Sample Program

```
//================================================================
// ABC to abc
//================================================================
// Description:
// In C-logic, a string variable can be viewed as an array of
// characters. This program demonstrates how you can modify a
// string using array notation.
//
// An Input string is searched for the occurrence of the
// uppercase string "ABC". If found, the uppercase "ABC" is
// replaced with a lowercase "abc" by replacing individual
// characters in the string.
//
//================================================================

public string Input;          // Input string
string s;                     // Local string variable
VAR index;                    // Array index

// Copy the input string to a local variable
s = Input;

// See if the input string contains an uppercase "ABC"
index = IndexOf(s, "ABC");
```

```
if(index >= 0)
{
      // Replace uppercase "ABC" with lowercase "abc"
      s[index] = 'a';
      s[index+1] = 'b';
      s[index+2] = 'c';
}
return s;
```

Maintenance Time Tracking Sample Program

```
//=================================================================
// Maintenance Time Tracking
//=================================================================
// Description:
// Most industrial machines require maintenance after a certain
// number of hours of operation. This program can be used to
// track the operation time, and schedule maintenance when the
// operation time reaches the maintenance due time.
//
// The program uses two inputs: a Reset and Input. When the Reset
// input is OFF and the Input is ON, the TotalTime accumulates
// the operation time. When the TotalTime reaches the
// WarningLevel, the Warning output is set. When the TotalTime
// reaches the MaintDueTime, the MaintDue output is set. When the
// Reset input is ON, all boolean outputs are turned OFF, and the
// TotalTime is reset to zero.
//
// Notes:
// This program demonstrates the use of the SetFixedInterval
// function, which allows scheduling of program execution at a
// fixed interval.
//
// In this implementation, the program's output is the same as
// the machine total time value (TotalTime). The program can
// easily be modified to return a different signal (e.g. MaintDue
// flag).
//
// To use a different data input, replace Item ID in the "ITEM
// Input" declaration.
//=================================================================

ITEM Input (".Input");                          // Data input

const WarningPercent(90.);                      // Warning level
as percent of the MaintDueTime
public double MaintDueTime(100.);               // Maintenance
due time in hours

public bool Reset(true);                        // Reset input
public bool Active(false,true,,,true);          // Active output
(Init false, Disable writes, Exclude from OnDataChange trig)
public bool Warning(false,true,,,true);         // ToolDue output
(Init false, Disable writes, Exclude from OnDataChange trig)
```

```
public bool MaintDue(false,true,,,true);        // MaintDue
output (Init false, Disable writes, Exclude from OnDataChange
trig)
public bool LastInput(false,true,,,true);       // The last state
of the input (Init false, Disable writes, Exclude from
OnDataChange trig)
public double TotalTime(0.0,,,,true);           // Total time
public double CycleStartTime(0.0,true,,,true);  // Current cycle
start time
public double CycleTime(0.0,true,,,true);       // Current cycle
time

double WarningLevel;
double TimeInHours;
var Temp;

// See if counter needs to be reset
if(Reset)
{
    // Reset
    LastInput = false;
    Active = false;
    MaintDue = false;
    Warning = false;
    TotalTime = 0.0;
    CycleStartTime = 0.0;
    CycleTime = 0;

    // Force recalculation of WarningLevel
    // when the counting starts
    WarningLevel.Quality = QUALITY_BAD;

    // Disable timer execution
    SetFixedInterval(0);

    return 0;
}

if(Input || LastInput)
{
    // Convert 100-nanosecond ticks to hours
    TimeInHours = TimeNow() / 36000000000.0;

    // See if Input transitioned from OFF to ON
    if(!LastInput)
    {
        // Save last input value
        LastInput = Input;

        CycleStartTime = TimeInHours;

        SetFixedInterval(100);
    }

    // Calculate current TotalTime
    Temp = TotalTime - CycleTime;
    CycleTime = TimeInHours - CycleStartTime;
    TotalTime = Temp + CycleTime;

    if(IsQualityBAD(WarningLevel))
        // Calculate warning level only once
        WarningLevel = WarningPercent * MaintDueTime / 100.0;

    // See if we exceeded the warning level
    if(TotalTime >= WarningLevel)
        Warning = true;

    // See if we exceeded the tool change time
    if(TotalTime >= MaintDueTime)
        MaintDue = true;

    // See if the cycle ended
    if(!Input)
    {
        // Save last input value
        LastInput = Input;

        // Disable timer execution
        SetFixedInterval(0);
        CycleTime = 0;
    }

    return TotalTime;
}

if(GetFixedInterval())
    // Disable timer execution
    SetFixedInterval(0);

// Don't change the output
return;
```

Rank Machine Performance Sample Program

```
//=================================================================
// Rank Machine Performance
//=================================================================
// Description:
// In a typical manufacturing operation, you may have a number of
// identical machines, and you may want to know which one of them
// is the slowest, and therefore requiring some extra attention
// or maintenance. Or maybe you are keeping track of machine
// cycles until maintenance due for multiple machines, and you
// want to know which machine will require maintenance first.
//
// This program demonstrates the use of the IndexSort function,
// which allows you to rank values in an array in order of
// highest to lowest or lowest to highest. In the example here,
// when the Hold input is low (false), the values in the
// InputArray are ranked from lowest to highest. When the Hold
// input is high (true), the last ranking is preserved, and the
// Holding output is turned On.
```

```
//
// Notes:
// To use a different input array, replace Item ID in the "ITEM
// InputArray" declaration.
//=================================================================

ITEM InputArray (".Input Array");                // Input array
public bool Hold(false);                         // Hold input
public bool Holding(false,true,,,true);          // Holding output
(Init false, Disable writes, Exclude from OnDataChange trig)

if(Hold)
{
    // Turn the Holding output ON
    Holding = true;

    // Do not change the output
    return;
}

if(Holding)
    // Turn the Holding output OFF
    Holding = false;

// Rank the input array
return IndexSort(InputArray);
```

Linear Conversion Sample Program

```
//=================================================================
// Linear Conversion
//=================================================================
// Notes:
// To use a different data input, replace Item ID in the "ITEM x"
// declaration.
//
// To change LowIR, HighIR, LowEU, HighEU, LowClamp, or
// HighClamp, modify the values assigned to these constants.
//=================================================================

ITEM x (".Input");              // Instrument data

const LowIR(0.0);               // Low instrument range
const HighIR(4095.0);           // High instrument range
const LowEU(0.0);               // Low engineering units range
const HighEU(100.0);            // High engineering units range
const LowClamp(0.0);            // Low clamping range
const HighClamp(100.0);         // High clamping range

double Result;                  // Conversion result

// Convert from instrument to EU range
Result = (x - LowIR)/(HighIR - LowIR)*(HighEU - LowEU) + LowEU;

// Check the clamping limits
```

```
if (Result < LowClamp)
{
    Result = LowClamp;

    // Set the limit field to Low Limited
    Result.Quality = QualityStatusCode(Result) |
QUALITY_LIMIT_LOW;
}
else if (Result > HighClamp)
{
    Result = HighClamp;

    // Set the limit field to High Limited
    Result.Quality = QualityStatusCode(Result) |
QUALITY_LIMIT_HIGH;
}
return Result;
```

Square Root Conversion Sample Program

```
//=================================================================
// Square Root Conversion
//=================================================================
// Notes:
// To use a different data input, replace Item ID in the "ITEM x"
// declaration.
//
// To change LowIR, HighIR, LowEU, HighEU, LowClamp, or
// HighClamp, modify the values assigned to these constants.
//=================================================================

ITEM x (".Input");             // Instrument data

const LowIR(0.0);              // Low instrument range
const HighIR(4095.0);          // High instrument range
const LowEU(0.0);              // Low engineering units range
const HighEU(100.0);           // High engineering units range
const LowClamp(0.0);           // Low clamping range
const HighClamp(100.0);        // High clamping range

double SqrtArg;                // Sqrt function argument
double Result;                 // Conversion result

SqrtArg = (x - LowIR)/(HighIR - LowIR);
if(SqrtArg < 0.0)
{
    Result.Quality = QUALITY_SENSOR_FAILURE;
    return Result;
}
// Convert from instrument to EU range
Result = sqrt(SqrtArg)*(HighEU - LowEU) + LowEU;

// Check the clamping limits
if (Result < LowClamp)
{
    Result = LowClamp;

    // Set the limit field to Low Limited
    Result.Quality = QualityStatusCode(Result) |
QUALITY_LIMIT_LOW;
}
else if (Result > HighClamp)
{
    Result = HighClamp;

    // Set the limit field to High Limited
    Result.Quality = QualityStatusCode(Result) |
QUALITY_LIMIT_HIGH;
}
return Result;
```

Array to Date & Time Sample Program

```
//=================================================================
// Array to Date & Time
//=================================================================
// Description:
// This program accepts the date and time information in the form
// of an array of date and time properties, and returns a string
// representing the provided date and time in the local locale.
//
// Notes:
//
//=================================================================

public VT_UI2 Array[8]({2010,1,1,1});     // Date & Time
properties array
DATETIME dt;

dt.Year = Array[0];
dt.Month = Array[1];
//dt.DayOfWeek = Array[2];          // Ignore the DayOfWeek (must
be valid, but the value is irrelevant)
dt.Day = Array[3];
dt.Hour = Array[4];
dt.Minute = Array[5];
dt.Second = Array[6];
dt.Milliseconds = Array[7];

return FormatDate(dt) + " " + FormatTime(dt);
```

Time in Your Time Zone #1 Sample Program

```
//=================================================================
// Time in Your Time Zone #1
//=================================================================
// Description:
// This program returns a string that represents the current date
// and time in the selected time zone.
//
// Notes:
// This program formats the output string by using the Format
// function with the date and time properties as arguments.
// Notice, that the resulting string does not depend on your
// system's locale (language) setting.
//=================================================================

DATETIME dt(UTC);                    // Current UTC time
// ZoneOffsetInHrs is public so that an external program can
// write the desired offset to it
public double ZoneOffsetInHrs(0.);  // Time zone offset in hours

// Read current time
dt = TimeNow();

// Add time zone offset
dt = AddSeconds(dt, ZoneOffsetInHrs * 3600.);

// Format the output string using the date & time properties
return Format("Your time: %d/%d/%d %d:%d:%d\n", dt.Month, dt.Day,
dt.Year, dt.Hour, dt.Minute, dt.Second);
```

Using Debug Outputs Sample Program

```
//=================================================================
// Using Debug Outputs
//=================================================================
// Description:
// This program demonstrates the use of the DebugOutput function
// as one of the debugging techniques.
//
// The DebugOutput function allows you to format an output string
// in the same way the Format function does. However, in addition
// to returning a formatted string, it writes the string to the
// selected debug output. The debug outputs are automatically
// created in the folder associated with the program's data item.
// The number of debug outputs depends on the number of
// OutputIndex values used in the program. The name of a debug
// output is "DebugOutputXX", where XX is a two-digit decimal
// representation of the corresponding OutputIndex. (The
// OutputIndex values 0-9 will have leading zeros appended to
// them.)
//
// While debugging your program, you place the DebugOutput
// function calls throughout your program. This allows you to
// monitor the state of internal variables, see which program
// paths are being executed, etc. Once you get your program
// working as expected, the DebugOutput calls can be removed or
// commented out.
//
// Notes:
//
//=================================================================
public double x(0.0);                // Input data
double y(0.0);                       // Local variable y

DebugOutput(1, "Old value = %f", x);

if(x >= 0.0)
{
    y = x * 100.0;
    DebugOutput(2, "New value = %f", y);
}
else
{
    y = x - 200;
    DebugOutput(3, "New value = %f", y);
}
return y;
```

Two Sines Sample Program

```
//=================================================================
// Two Sines
//=================================================================
// Mathematically, the sine function takes values between -1 and
// +1. This means that the value will vary both above and below
// the level set by the offset. Therefore, this is the only
// function for which the Offset parameter specifies the middle
// of the range, rather than the bottom. In addition, this is the
// only function for which the Amplitude parameter does not
// specify the peak-to-peak range of values.
//
// Notes:
// This program is similar to the Sine program, but it generates
// two sine waves shifted in phase. The output of the first sine
// (SineA) is sent to this data item's output, while the second
// sine wave is sent to the public variable called SineB.
//
// To change Offset, Amplitude, Period, or Phase, modify the
// values assigned to these constants.
//=================================================================

const OffsetA(0.0);           // Fixed offset for Sine A waveform
const OffsetB(0.0);           // Fixed offset for Sine B waveform
const AmplitudeA(100.0);      // Peak value for Sine A, measured
from the offset level
const AmplitudeB(100.0);      // Peak value for Sine B, measured
from the offset level
const PhaseA(0.0);      // Phase shift in radians for Sine A
const PhaseB(1.57079632679489661923132169163 95);// Phase shift in
radians for Sine B (pi/2.0)
const Period(100.0);                 // Period in seconds

public double SineB(,,,,true);       // Sine B output
double TimeInSeconds;                // Time in seconds
double AngleBase;
```

```
// Convert 100-nanosecond ticks to seconds
TimeInSeconds = TimeNow() / 10000000.0;
AngleBase = TimeInSeconds*2.0*PI/Period;

SineB = OffsetB + sin(AngleBase + PhaseB)*AmplitudeB;
return OffsetA + sin(AngleBase + PhaseA)*AmplitudeA;
```